Performance improvement of distributed key-value store using van emde boas

Shreeya Deshpande,Varsha Jha, Affan Shaikh, Niket Doke, Aparna Mete-Sawant

Abstract— Distributed Key-Value store systems are relevant and of great benefits in almost all systems that aim at providing services to a considerably large amount of audience which is quite spread out geographically. All major conglomerate firms are highly dependent on distributed key-value store and even the slightest failure of the system could result in losses that are not only monetary in nature but are also in the form of highly sensitive information that could alter the functioning of a great deal of highly sophisticated systems. So we might safely say that the slightest of deviation from the norm in the functioning of distributed key-store value could result in grave consequences for all the stakeholders involved. One of the most straightforward implementation of such a system could be by using a hash table to store all the key-value pairs. Such an implementation is obviously quite easy to use and would also allow us to read/write key-value pairs in constant time but it also has an obvious downside. When we use a hash table we need to store all the data available in memory which is quite difficult if the data is too big which is mostly the case. Instead of using a hash function this paper tries to implement a concept called consistency hashing. Also instead of using plain vanilla consistency hashing this paper tries to implement a slightly modified version that introduces some variation to tackle some of the drawbacks. This paper tries to minimize the trade-offs that are usually made in the development of such systems thereby making it reliable, available and consistent at the same time. The parameters that the paper majorly focuses on are performance, reliability and scalability of the system. The main question that arises before building such a system is how to distribute the data on multiple machines and what is the most optimal method or strategy to partition the data. Sharding and the modified consistency hashing most fittingly answers these questions. While scaling such a system a lot of other factors come into play like hardware support, replication requirements, indexing, read/write volume, query pattern, size of value and complete data, caching, vector clocks, gossip protocols, repair strategies, etc. this paper attempts to tackle each of these factors meticulously. However the main concern of this paper remains the latency or the response time of the system. It tries to minimize the latency without compromising other trade-offs or factors involved.

Index Terms – Key-value, consistent hashing, replication, sharding, partitioning, load balancing, van-emde boas

1. INTRODUCTION

Any system that aims at providing services to a considerably large amount of audience at its peak time would expect to address a crowd of up to but not limited to about tens of millions. To provide services to that amount of customers one would require thousands of servers located globally. Any firm addressing and providing services to a large crowd would most obviously have a strict set of requirements as to on which factors does it want to focus the most. The factors could be performance, reliability, availability, efficiency, etc. To ensure that the system is able to adapt the continuous and rapid growth of the number of customers using it the system must also be scalable in all its aspects. In this paper reliability and availability are some of the factors that are majorly focused on as even the slightest of the downtime of the system could have grave consequences monetarily and also in terms of customers' trust on the platform or system. Scalability is another factor that this paper takes into consideration as there is no point in having a system in place that cannot entertain and efficiently handle growing number of customers on a regular basis. All the factors mentioned above largely depend on how the state of the platform or system is managed. This paper attempts to implement a system that is highly dispersed i.e. decentralized and is loosely coupled and can support multiple nodes and services or operations. In today's world there is a growing and urgent need of technologies and platforms that are available 24x7. For example, the customer must be able to perform operations and avail services even if the system is crashing or failing, the networks are failing or the servers which are geographically spread out are facing earthquakes or tornados or tsunamis. For this to be possible the factors responsible to manage the availability of the system must always be able to read from and write to all the databases that are spread out globally.

Recovering from failure or crashing of a system that comprises of multiple nodes and components is a precarious and delicate situation. This paper tries to handle such situation assuming that crashing or failing is the general mode of operation of the system without giving it a separate sophisticated mechanism. This paper tries to recover from failure without affecting the performance and reliability of the system and causing minimal effect on the functioning of the system.

This paper tries to propose the implementation of a highly reliable, available and scalable distributed key-value store.The proposed system is highly inspired by the already implemented systems like Amazon's Dynamo and LinkedIn's Project Voldemort. Such a system is used to efficiently handle and manage the operations that have high reliability and availability requirements. At the same time it tries to minimize the trade-offs of not only between consistency and availability but also between performance, cost effectiveness, efficiency, speed and feasibility. To handle the disparate operations that the systems offer today we need to have a storage mechanism that is malleable enough to let the developer manage the configuration of the overall distributed system based on the trade-offs and the availability and performance parameter the developer wants to achieve. Also the storage technology needs to be feasible in terms of cost. This paper implements such a

system by integration of a variety of well-known techniques and tweaking parameters wherever needed, to fit the need of the system of achieving high availability and performance threshold. The data which is going to be stored is segregated and cloned using a variation of consistency hashing which is aided and expedited by the concept of versioning. The consistency of data which is partitioned and replicated across multiple nodes is maintained by a quorum based mechanism and a special synchronization protocol that maintains consistent replicas of the data in question. However the bottom line of this paper is that the amalgamation and proper synthesis of certain well-known techniques with a little variation could result in the development of a system that minimizes the compromises and trade-offs among the important factors and achieves a highly available and efficient system.

2. LITERATURE SURVEY

2.1 Related Work

In the paper [1] Giuseppe DeCandia et.al try to explain the design of a distributed key-value storage which is non-relational in nature created solely to handle and cater to the services of the customers of Amazon. These services might include list of best selling items on the website, management of shopping carts, management of the session of a particular customer, etc. The name of this system is Dynamo. Dynamo has no single point of failure as its architecture which is quite loosely coupled and dispersed in nature runs on top of a network which is peer to peer structured. Every node in the network is well aware of every other node present in the same network. To provide availability, reliability and durability Dynamo stores replica of a particular unit of data over multiple nodes in the network. Each node present in the network has the responsibility of the data falling in a particular key-range. The node also called as the coordinator has a list of where and how it would prefer to replicate the data and store it. Dynamo is quite scalable and can at its peak time handle thousands of nodes across multiple databases. On shortcoming of Dynamo is that it highly compromises consistency of data across multiple replicas to provide and ensure no downtime or availability so that the users have good accessibility. The main aim of Dynamo is to always make the readable/writable data available so that the user request to read or write data is never rejected even I the data centers are facing tornados, earthquakes or tsunamis. This is what gives rise to inconsistencies of data among multiple nodes. At a given time it might be possible that the data corresponding to a particular key might not be consistent or same across two different replicas of the data. To handle this Dynamo uses a mechanism in which the data across all the replicas are eventually made consistent in the background by gradual tweaking. When the user avails the update() function Dynamo does not overwrite the existing data but instead uses a mechanism called data versioning to store the previous copies of data. Dynamo uses vector clocks to resolve the conflict between multiple versions of the data with the same key. If any operation that the user wants to perform fails or is not able to completely take place and the coordinator fails to receive any acknowledgement then there is no recovery operation in place to retransmit the operation i.e. there is no timeout-retransmission discipline in place.

In the paper [2] Roshan Sumbaly et.al tried to implement another distributed key-value store called Voldemort which was initially developed to cater the needs of LinkedIn users but was later (in 2009) open sourced for any developer to use, modify and implement in their own project. Project Voldemort is highly driven by the concepts used in Dynamo. For example like Dynamo its architecture runs on top of a network which is peer to peer in nature. There is no master and each node is equally responsible for any task that comes to the system. To provide availability, reliability and durability Voldemort like Dynamo stores replica of a particular unit of data over multiple nodes in the network. For this purpose it uses consistent hashing and partitions the data accordingly. As mentioned above each node is quite independent and does not depend on any other node for its functioning, there is no single point failure. Voldemort has a layered architecture where each logical layer has its own set of operations and responsibilities to perform. Since it does not provide the ACID properties it is not relational. Storage formats like MySQL and BerkeleyDB are supported by Voldemort. Addition and removal of nodes can take place in the Voldemort network without affecting the normal functioning of the system. These changes are automatically detected by the recovery mechanism of Voldemort. Like Dynamo. Voldemort compromises consistency for availability. Any inconsistency among data replicas are solved in the background while the read operation is taking place. This is called read repair. Sometimes read-repair can be too expensive. At such times, a mechanism called hinted handoff is used to resolve the conflict. Voldemort has no timeout-retransmission discipline in place just like Dynamo.

In the paper[3], Aimen Mukhtar et.al tried to implement another NoSQL key-value data store called Riak which is highly inspired by the concepts of Dynamo. Like Dynamo its architecture runs on top of a network which is peer to peer in nature. In Riak multiple users can update the data concurrently at the same time and the conflicts in data versions are resolved in the background by read repair. Riak also uses vector clocks and hinted handoff to resolve the conflicts that arise due to the concurrent writes. Riak has no single point of failure as its architecture works in top of a ring topology. Riak makes sure that either a success or a failure message is received by a node requesting an operation i.e. it makes sure that acknowledgement for every operation is received by the requesting node hence there is a timeout-retransmission discipline in place which kicks in anytime a failure occurs.

In this paper [4], Avinash Lakshman tries to explain the implementation of a column-oriented data store named Cassandra. Cassandra was initially developed exclusively for Facebook but was later open sourced in 2008. Cassandra is designed to manage a large amount of data. It is fully dispersed and implements a peer to peer ring topology. It integrates the features of Google's Big Table and Amazon's Dynamo providing excellent features to store data extensively in rows and columns. In Cassandra the data which is to be stored is partitioned both horizontally and vertically using consistent hashing and also cloned over multiple nodes in the network to perform load balancing. The ring topology of Cassandra uses gossip protocols for communication between the nodes. The purpose of such communication is to exchange messages about the state of the application and also to alert all the nodes about any kind of failure that may occur. This helps in the recovery of the system after a failure. In Cassandra the client decides the level of consistency to be maintained among the replicas of the data. Cassandra does not fully provide all of the ACID properties. Whenever a conflict occurs, the timestamp mechanism is used to resolve it.
In this paper [5], Fay Chang et.al tried to explain the design of Big Table which is another NoSQL data storage used in a number of Google applications. Big Table maintains three attributes for each of the key-value data pair in the database: first a unique key keeps the name of the URL in reverse; second a unique column that keeps the entire contents of the webpage that the URL is pointing to and also some additional keys to store references to the URL pointed web page; third is the timestamp which denotes at what time the data was asked for or referred to. All the three attributes are stored by the Big table as strings. In Big Table, data is stored in a dictionary like format. Column families in Big Table are groups of columns that have data of same data type. This drastically reduces the number of columns present in a table. Column families are created before any data is stored. B+ trees are used to store rows in Big Table. Big table has single point of failure since it is highly dependent on GFS or master based architecture which can become a bottleneck for some operations.

In this paper [6], Hiren Patel et.al tried to explain the implementation of another NoSQL, column-oriented data storage technology which is developed on top of the Hadoop architecture named HBase. It is quite similar to Google's Big table with the added feature of being open source. HBase can handle billions of data entries with millions of features. It uses data storage mechanisms quite similar to Cassandra with the exception that HDFS is used to store data. HBase has a quite complex and elaborate master-slave architecture hence can have single point of failure. The master is called HMaster and the slaves are called region servers (RS). Similar to Cassandra there are column families in HBase to significantly reduce the number of columns. In case of a failure it is very hard to recover for HBase hence it is not quite scalable. It has strong mechanisms in place to manage consistencies. Also it has timeout-retransmission discipline in place for receiving acknowledgement

3. SYSTEM ARCHITECTURE

3.1. System Architecture

Highly inspired from Voldemort, there are two possible architectures: 3 tier and 2 tier. In the 3 tier architecture, the client has no information about routing and forwards the request to the load balancer which then forwards the request to the backend. The backend has all the necessary routing information (partition aware routing) and finally forwards the request to the dedicated server. Thus, in a 3 tier architecture every request needs 2 hops for execution which increases the response time but, it also makes the client lighter and independent of the keystore implementation.
On the other hand, in the 2 tier architecture, the client is aware of the partition aware routing and forwards the request to the dedicated server directly making it much faster. However, in order to get the routing information the client needs to perform bootstrapping
process and needs to keep in sync with the system. Both the architecture are possible but for testing the system, 3 tier architecture is used.

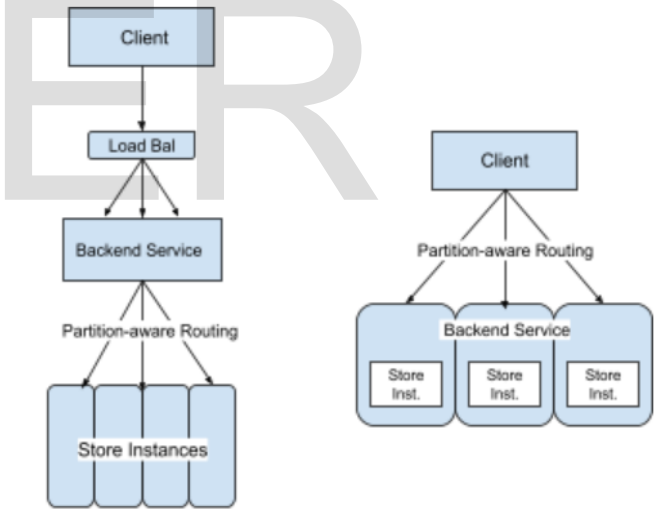


Figure 1 : System architecture : a) 3 tier Backend Routed b) 2 tier Client Routed

3.2. Logical Architecture

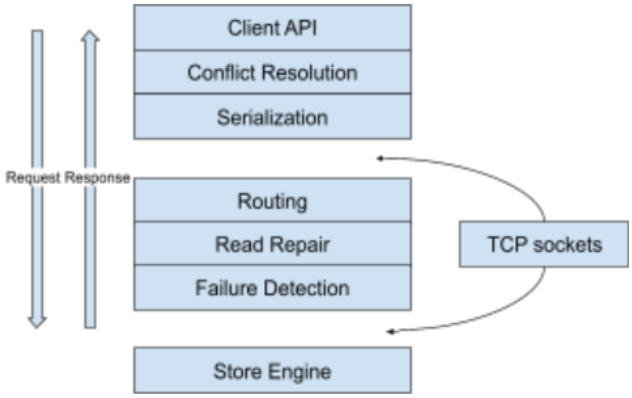


Figure 2 : Logical Architecture

The proposed architecture is a peer to peer architecture consisting of symmetric nodes. Each of the nodes consists of different logical modules responsible for different features. Every node has routing information and information about every other node in the system. The system supports storing data in the form of key value form among different nodes. The system can be configured using N (replication factor), W (write count) and R (read count). N is the replication factor indicating the number of nodes the data(key value pair) is to be replicated. W (write count) indicates the number of node's responses the system blocks before declaring write successful. R (read count) indicates the number of parallel gets before declaring a successful get.

Elaborating the modules, at the top is client api, which can be used for sending requests. The proposed system supports get, put and delete operations. The client can or cannot have routing information depending on the configuration. As it's a distributed system, it's inexorable to have data inconsistency. There's where the conflict resolution module comes into picture which handles this inconsistency based on the setting. The key value store returns all versions of data for a given key to the client which can further apply filters like choosing the one with the latest timestamp or taking union of them based on the use case. The serialization module is responsible for serializing the data before sending it through the network and deserializing the received data from the request. The system uses json for serialization and deserialization.

The routing model is responsible for both partitioning and balancing the load among the nodes, The routing is done by a slightly modified version of consistent hashing with different data structure. It also handles node addition and deletions. There's also a module for dealing with conflict resolution called the read repair module. The failure detection module looks for availability of nodes and shares the status with everyone in the network. Finally, the storage engine is used for storing the data in the form of a key value store. Any database can be used here, the system uses Berkeley DB.

.

3.3. PARTITIONING

3.3.1. Consistent Hashing

One of the most basic requirements of the proposed system to be scalable is that it should be able to add or remove nodes in the network and accordingly manage the distribution of data over the new set of nodes and the entire process must be dynamic. Consistent Hashing can be the solution to achieve this task. In consistent hashing whichever hash function is being used, its output range is treated as a circular ring which is fixed. Due to the circular ring-like nature the largest output value of the hash function is wrapped up and mapped to the smallest hash value. Each node in the network is first given a "name" which is a random string or value. This random value decides the position of the node on the ring discussed above. Then the data corresponding to each key is passed through the hash function and the output value then decides on which node the data will be stored. We do so by going around the ring in clockwise direction and finding the first placed node on the ring with position value larger than that yielded by the hash function for the data. In this way each node on the ring is only responsible for the data that falls in the region between itself and the node before it. The primary advantage of consistent hashing is that whenever a node is added or removed from the network, only the node in the neighborhood of the node in question are affected leaving the rest of the network unaffected and intact. However, there are some challenges that such an approach neglects. Firstly the randomness in assigning positions to the nodes in the ring causes unbalanced load distribution. Inorder to deal with uneven load balancing virtual nodes are being added for each node in the network. Secondly, this approach pays no attention to the diverseness of the nature and performance threshold of the various nodes in the network. To overcome some of these challenges, this paper uses a modified version of consistent hashing.

3.3.2. Consistent Hashing Implementation

Generally, Consistent hashing is implemented by using any balanced key value tree structure. Hence, the operations like finding a node associated with a key has a time complexity of log N, where N is the number of entries in the structure (number of nodes + number of virtual nodes). This time complexity is still very good but can be improved by using a data structure called Van Emde Boas Tree or VEB invented Dutch computer scientist Peter van Emde Boas in 1975. It supports insert, successor, predecessor and delete operations in log log M. where M is range of key in the structure. In case of consistent hashing, M can be the range of the hashing function used. Hence, using a VEB tree can improve the time complexity of finding node for a particular key exponentially. Considering the output of the hash function is m bit number then using a VEB tree time complexity will be log m (log 2m is m) which is much lesser than log N. Moreover, the

complexity is independent of the number of nodes and hence, the system can scale better.

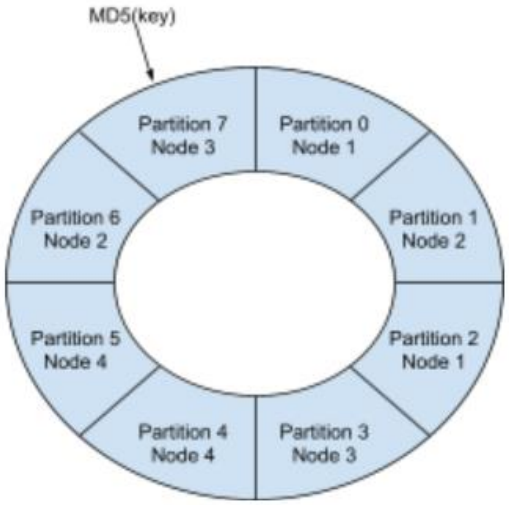


Figure 3 : Ring topology for number of nodes N = 4 and number of partitions Q=8. A key whose hash falls in the range of partition will be handled by node 3.

Moreover, normally the nodes are placed on the ring randomly using the hashing function. However, in this implementation, the ring is divided into Q equal partitions and these partitions are then randomly distributed among the nodes N. So, each node corresponds to Q/N partition. When a node is deleted, the partition corresponding to that node are distributed among the other nodes randomly. Similarly, whenever a new node is added some of the partitions are chosen randomly from the previous nodes and are given to the new node. Using this technique ensure always equal load distribution among the nodes.

3.4. Replication

To ensure that there is no down-time and the system is available all the time, the data that is to be stored is replicated across multiple nodes in the network. Each key-value pair is replicated across N nodes where N is basically a parameter decided by the developer. From each key-value pair the key k is taken and is assigned to node called the coordinator. Each coordinator is responsible to replicate the data objects that lie within its range. Apart from storing all the data objects of the keys that lie in its range locally, the coordinator also clones these data objects and stores them at N-1 nodes that are located after its position, clockwise on the ring-like structure of consistent hashing. In this way each node on the ring is only responsible for the data that falls in the region between itself and the Nth node preceding it. The groups of nodes storing the replica of any data object are individually called the "preference list" for that particular key k whose data object they are storing. Each node in the network has equal rights and responsibility to decide that for a particular key which all nodes should be present in the preference list. To handle system failures the preference list may contain more than N nodes for a particular key.

3.5. DATA VERSIONING

In case on any inconsistency among the replicas of data, the conflict is resolved in the background asynchronously. There may be a case where the user initiates a put() operation. Concurrently another user might want to update some entries using the put() operation. It may happen that some replicas of the data might not be able to update the key-value pair and hence a subsequent get() operation may not return the latest key-value pair. If there is no failure than the eventual consistency mechanism propagates the updates within a given period of time but if a failure like system crash occurs then the propagation of the update may be postponed indefinitely. In order to deal with this situation, data versioning is used. In data versioning, even if the data is modified or updated, the older versions of the data are still stored. Each version of the data is immutable and can be retrieved whenever needed. Therefore, multiple data objects of the same key value are present within the system. Many a times, the newer version of a data object may absorb the previous versions. At such times the system decides which version has the authority to do so. But at the time of system failure combined with concurrent requests to update a data object, the client has to decide which version would subsume which of the existing versions and which versions need to be preserved. Merging of versions may also take place and also the deleted versions may reappear. The key point here is that system failure may result in more than two version of the same data object. These replicas may later reconcile or subsume each other. We need some sophisticated discipline that acknowledges that there may be more than just two version of a particular data object.

Vector clocks are used to trace out the provenance of connection between any versions of the same data object. A vector clock is basically a numbered list of pairs of node and counters. Each version of each data object has a vector clock associated with it. By simply looking at the vector clock one can determine whether two versions are on parallel branches or are on some cause-effect branch. Consider two version of the same data object. If the counters of the first version's clock are all either less than or equal to all the nodes part of the second version's clock then the first version is called the ancestor and the second version is deleted. Otherwise there needs to be some conflict resolution and subsuming of versions. Whenever a client wants to update a data object it needs to mention which version it want to update by giving out the context in terms of the vector clock details. If the system then has access to multiple branches that cannot be logically subsumed or merged then it returns all the data objects at the leaves with the corresponding versions and

context in the form of vector clocks. Using this context the divergent branches are then merged into a single version.

3.6 API

The proposed system stores the key-value pair based on a quite simple and straightforward principle. It supports three operations namely get(), put() and delete(). The put() operation can also be used to update the data objects. The get() function take the key as the parameter and returns the value associated with the key. It is also possible that the get() operation might return a list of data objects with different versions that might be conflicting along with the context of each version in terms of the respective vector clocks. The put() operation takes the key, context and object as the parameter and decides where the data corresponding to the key is to be written and accordingly writes to each of the replica nodes. The context is used to maintain the versioning. The context is not quite visible to the caller of the operation. It is internally used to resolve conflicts among the replicas of the data object.

3.6.1. Data Model

The key store in the proposed system only supports string data type for both keys and values. The following table explains the data constraints.

TABLE 1
TABLE DESCRIBING THE TYPE OF DATA THAT CAN BE STORED IN THE SYSTEM.

Feature	Supported Data Type	Size
Key	String	512
Value	String	1024

3.6.2. Execution

All the nodes in the network are capable of receiving get(), put() and delete() requests from the client side. Here we are assuming that the environment is failure-free. All the operations supported by the system are invoked over HTTP. The operation requested by the client can be carried out in two ways. The first is whenever the request arrives, it is first forwarded to a coordinator that tries to balance the load and based on the information about load distribution in the network assign a node in the network to carry out the operation. In the second scenario the client itself is well aware of the load distribution among the nodes and directly forwards the request to the designated node. The advantage of the first method is that no additional code is to be written to make the client self-aware. On the other hand, the second approach may result in less response time as a potential time-consuming step of forwarding the request is eliminated. A node that is given the responsibility of handling the read/write operations is called the coordinator. If the node that will become the coordinator is selected by the first approach mentioned above, it will most probably be first in the preference list of the said key consisting of top N nodes. If that is not the case, i.e. the coordinator is not present in the preference list of the key, then the present coordinator will relinquish the control and forward the request of the said operation to one of the nodes that is present in the preference list. Usually, the top N nodes in the preference list are considered while deciding the coordinator. However, in case of system crash or failure, the lower-lying nodes in the preference list may also be considered to carry out the requested operation.

To resolve any conflicts arising due to inconsistencies among replicas, the proposed system uses a quorum-based protocol system to maintain consistency among data objects. This protocol has two parameters R and W. R is the minimum number of nodes that need to be read from in order for a successful read operation. W is the minimum number of nodes that need to be written to for a successful write operation. R and W both are set such that R + W > N. Here the response time of a get() operation is decided by the slowest R clones and the response time of put() operation is decided by the slowest of W clones. R and W both are hence usually kept less than N. When a put() operation arrives, the system first generates a local set of context consisting of the vector clock and a new version of the said key. This version along with the context is then sent to the top N reachable nodes in the network. If this new version is successfully written on W-1 top nodes in the network then we consider the put() operation to be successfully executed. For a get() operation the coordinator gathers all the versions of the data object corresponding to the said key. It returns the result of the operation to the client after receiving R responses from the network. If there are conflicting multiple versions of the same data object then based on the context i.e. vector clocks the versions are reconciled and then returned to the client.

3.7. HANDLING FAILURE

Handling failure is quite important so that each node is aware that which nodes in the system are currently down and the node that is up does not try to communicate with a node which is currently down. If a strict quorum-based protocol is used then the system may not be handle even the simplest of the failure. So to overcome this hinted handoff is used. While an operation is requested by the client only the first N reachable nodes are considered. These nodes need not necessarily be the top N nodes while going round the ring of consistent hashing. For example, consider a situation where a data object is to be written on node A. But due to some reasons the node A is currently down. Using the hinted handoff discipline the system writes the data object on another node which is currently up. Let this node be B. Node B stores the information that the data object written to it was intended to be written on

node A. Once the node A is up again, the data object is copied from B to A and node B deletes the data object from its memory. This ensures that the system is available all the time, even in the case of a system failure. To ensure that the system accepts a write operation at all costs as long as there is at least a single node present to write the data object to itself the developer can set W to 1. However, to maintain a higher durability level it is better to set the value of W a bit higher.

3.8. MEMBERSHIP AND FAILURE DETECTION

3.8.1. Ring Membership

The down time of any node due to system failure is usually ephemeral but may sometimes last for a long period of time. A node being down for a while does not imply that the node has permanently departed from the system neither does addition of a node to the network due to some manual error form the client side imply that the node has been permanently added to the system. Hence in such situations, the load must not be completely re-structured. To handle this, the proposed system has a separate sophisticated mechanism in place to add or remove a node from the network. The admin of the system uses a command line tool to make a node at random in-charge of issuing a membership change while adding or removing a node from the network. This membership change issuing node stores the changes and the time of change in a perpetual store. These changes together form a history. This is because the nodes are continuously being added and removed from the network. All the nodes in the network continuously keep communicating with each other and hence they use a gossip-based protocol to keep the membership changes intact and consistent. The nodes communicate with each other and eventually keep their membership changes histories consistent. The details regarding partitioning and load balancing is also propagated using the gossip protocols.

3.8.2. External Discovery

The mechanism discussed above for adding or removing a node to the network may lead to a logically partitioned network where the nodes are not aware of each other's presence. For example, if admin adds a node A in the ring of the network. At the same time admin adds another node B to the ring of the network. Neither of them would be aware of each other's presence for quite some time. This would result in logical partitioning of the network. To handle this the concept of coordinator nodes are used. These coordinator nodes are nodes in the network whose presence is known to all at all times. Any node being added to the network eventually merges its membership with these coordinator nodes. These coordinator nodes like any other node in the network can perform all the operations offered by the system. These coordinator nodes are different from the coordinators that decide the replication factor of a data object as discussed in above sections.

3.8.3. Failure Detection

The proposed system handles failure so that each node is aware that which nodes in the system are currently down and the node that is up does not try to communicate with a node which is currently down. For example, a node A considers its peer node B to be down or failed if node b is not able to respond to node A's messages in a stipulated period of time. The node A then uses alternate nodes in the network that are capable of together performing the operations that node B could perform alone. Meanwhile, node A also continues to check whether node B is up and running again or is still down by sending node B messages. Initially, gossip protocols were used by the system to maintain a global state which is consistent of all the failed nodes. But later it was observed that this maintenance of a globally consistent list is not really necessary. This is because since the nodes continuously join and remove the network, by gossip protocols all the nodes are made self-aware of the state of the entire system. There is no need of explicitly acknowledging the failure of a node by making a list. Another way of detecting failure is by Phi Accrual Failure Detection Mechanism. In this method, we don't have a binary answer as to whether a node is down or is up and working. Instead we get a suspicion value of whether the node is down or not. This suspicion value is expressed as φ. This φ is described on a scale which can be adjusted dynamically according to the load distribution on the nodes being monitored for failure detection. The basic idea is that we first set a threshold for φ. Then we suspect a node say node B. Now if φ=1 then the probability of us making a mistake in the suspicion of the node B is 10%. Similarly the probability of making a mistake is 1% if φ=2, 0.1% if φ=3, and so on. Each node in the network maintains the time interval of receiving messages in the operation of gossip protocols. These intervals are considered while calculating φ. This method works well for Exponential Distribution for approximation purposes. Also this method is quite efficient, fast and accurate considering the load on the nodes on the network.

4. EXPERIMENTATION AND EVALUATION

This section briefs about the response time and other latency measurements of a prototype implementation of the proposed system. The evaluation metrics that are proposed in the implementation of the said system answers the following questions: -

Is the system infinitely scalable?

What is the response time for get () operation and does it reduce with the addition of new nodes?

What is the response time for put () operation and does it reduce with the addition of new nodes?

Is the system continuously available in a given interval of time?
At its peak time, how many requests can the system handle?
At its peak time, how many requests can the system successfully complete?

After the implementation of the prototype of the system it was found out that the system is significantly reliable and available (almost 99.98% of the time). Also, the requests for operations were successfully completed (almost 99.997% of the time) without any major glitches and no data was lost due to any error or failure of the system. Also the response time significantly reduces (by almost 20%) by adding a new node to the system which makes it infinitely scalable.

5. RESULTS AND ANALYSIS

This section explains the results obtained and also the analysis performed on these results. The system was tested with 3, 6 and 9 servers respectively. The servers were loaded with 150 clients simultaneously. A threshold(T) indicating the maximum response time is obtained for each specific case. It is guaranteed that the system does not cross this threshold for any number of clients.

Number of Servers	Response Time Threshold for Put (R_p) in ms	Response Time Threshold for Get (R_g)
3	4858.33	2533.35
6	4912.72	3776.47
9	4812.87	2485.52

Table 2 : Number of servers vs Threshold for Put(R_p) and Get(R_g) respectively.

From table 2. the system guarantees that for 3 servers. the put request won't ever cross 4858.33 ms and the get request won't ever cross 2533.35 ms, for 6 servers. the put request won't ever cross 4912.33 ms and the get request won't ever cross 3766.47 ms and for 9 servers. the put request won't ever cross 4812.87 ms and the get request won't ever cross 2485.52 ms.

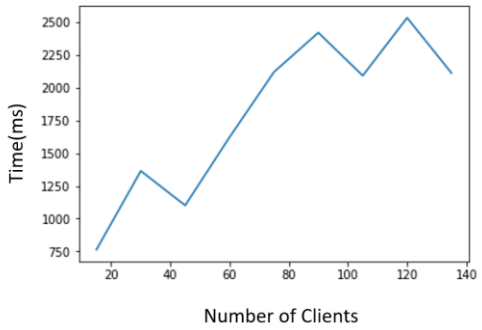


Figure 4. 3 Servers (Get Response time vs Number of clients)

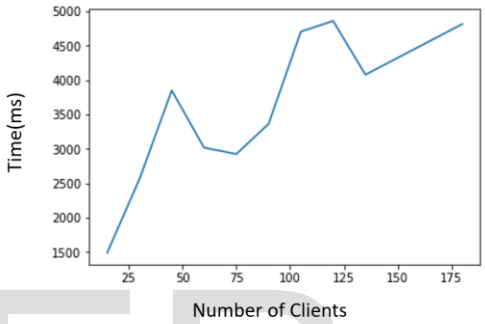


Figure 5. 3 Servers (Put Response time vs Number of clients)

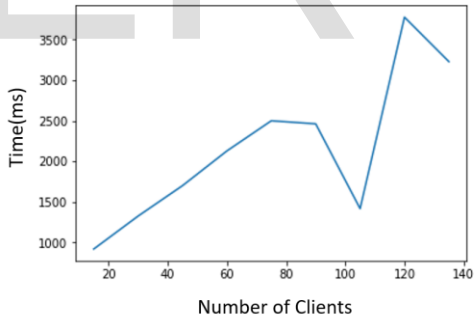


Figure 6. 6 Servers (Get Response time vs Number of clients)

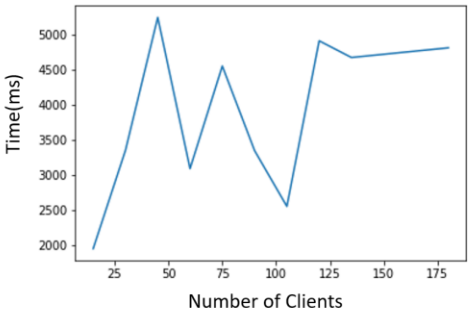


Figure 7. 6 Servers (Put Response time vs Number of clients)

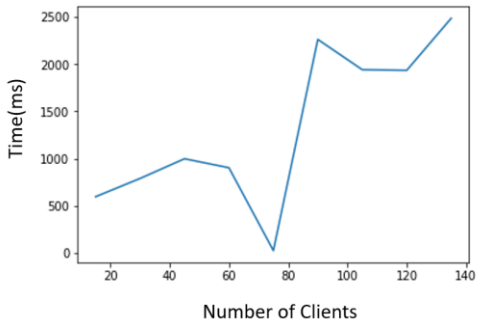


Figure 8. 9 Servers (Get Response time vs Number of clients)

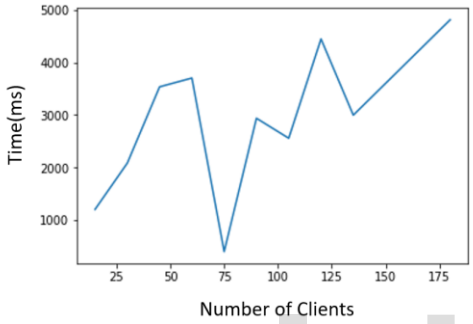


Figure 9. 9 Servers (Put Response time vs Number of clients)

In all of the above experiments the load on the servers are gradually increased by simultaneous loading. The simultaneous loads are initially started from 15 clients and taken all the way up-to 140 clients. As the number of clients are increasing the response time increases for put operation. Similar to this for the get operation also the response time increases. For both the operations the response time has a specific threshold as discussed in Table 2.

6. CONCLUSION

This paper described the design and implementation of a distributed key-value store which is highly available, scalable and durable. The system was successful in achieving the desired levels of performance and was able to handle all types of failures and system crashes and all types of partitions. The proposed system's latency decreases with increase in the number of nodes in the network and hence it is infinitely scalable. According to the users' need, the system can be scaled down or up. By tweaking the parameters N, R and W the desired levels of scalability, performance and availability can be met.

7. FUTURE SCOPE

In the proposed system many decentralized techniques are used. All these techniques can be integrated with the appropriate parameters and can be used to make a single highly reliable and available system. A way to reduce the inter-node communication latency is to extract a single data object from HDFS and then propagate it among other nodes in the network. Also Hadoop could be used to build techniques to increase the speed of the lookup process.

8. REFERENCES

[1] Giuseppe DeCandia et al. "Dynamo: Amazon's Highly Available Key-value Store", 2007.
[2] Manjula Suresh et al. "Serving Large-scale Batch Computed Data with Project Voldemort", 2009
[3] Aimen Mukhta et al. "Evaluating Riak Key Value Cluster for Big Data" 2020
[4] Avinash Lakshman et al. "Cassandra - A Decentralized Structured Storage System", 2009
[5] Fay Chang et al. " Bigtable: A Distributed Storage System for Structured Data",2006
[6] Hiren Patel et al. " HBase: A NoSQL Database " 2017
[7] Yasin Celik "A Study on Scalability of Distributed Key-Value Pair Systems",2016.
[8] Tuncay Bayrak, "Performance Metrics for disaster Monitoring Systems"
[9] P.Basu, W.Ke and T.D.C Little,"Metrics for Performance Evaluation of Distributed Application Execution in Ubiquitous Computing Environments"
[10] Lada Adamic and Bernardo Huberman. Zipf's law and the Internet. Glottometrics, 3:143–150, 2002
[11] Antonio Barbuzzi, Pietro Michiardi, Ernst Biersack, and Gennaro Boggia. Parallel Bulk Insertion for Large-scale Analytics Applications. In Proceedings of the 4th International Workshop on Large Scale Distributed Systems and Middleware (LADIS '10), pages 27–31, New York, NY, USA, 2010.
[12] Balaraja Subbiah, "Lamport Clock and Vector Clock" at https://medium.com/@balrajasubbiah/lamport-clocks-and-vector-clocks-b713db1890d7
[13] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson Hsieh, Deborah Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert Gruber. Bigtable: A Distributed Storage System for Structured Data. In Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI '06), Berkeley, CA, USA, 2006
[14] Brian Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, HansArno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. PNUTS: Yahoo!'s Hosted Data Serving Platform. Proceedings of the VLDB Endowment, 1:1277–1288, August 2008.
[15] Brian Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC '10), pages 143–154, New York, NY, USA, 2010
[16] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. In Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation – Volume 6 (OSDI '04), Berkeley, CA, USA, 2004.
[17] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan

Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-Value Store. SIGOPS Operating Systems Review, 41:205–220, October 2007.

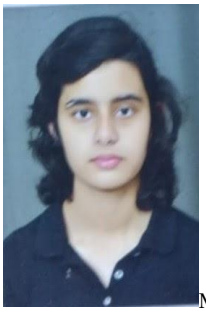
[18] M. Dowell and P. Jarratt. The Pegasus method for computing the root of an equation. BIT Numerical Mathematics, 12:503–508, 1972.

[19] Lars George. HBase: The Definitive Guide. O'Reilly Media, 2011.

[20] Ioannis Konstantinou, Evangelos Angelou, Dimitrios Tsoumakos, and Nectarios Koziris. Distributed Indexing of Web Scale Datasets for the Cloud. In Proceedings of the 2010 Workshop on Massive Data Analytics on the Cloud (MDAC '10), pages 1:1–1:6, New York, NY, USA, 2010.


Ms. Shreeya Deshpande (Research Scholar)

She is a student of BTech in Information Technology from Vishwakarma Institute of Technology. She has completed an internship at Nvidia as a System Software Engineering Intern. Her interests include Artificial Intelligence, Deep Learning, Machine Learning and Distributed Systems.


Ms. Varsha Jha (Research Scholar)

She is a student of BTech in Information Technology from Vishwakarma Institute of Technology. She has completed an in-house internship at Vishwakarma Institute of Technology in Internet of Things and has also done a semester long internship at Symantec. Her interests are Blockchain, Internet of Things and Machine Learning. She has published papers in the area of IoT and Blockchain


Mr. Affan Shaikh (Research Scholar)

He is a student of BTech in Information Technology from Vishwakarma Institute of Technology. He has completed an internship at Credit Suisse. His interests include Natural Language Processing, Machine Learning and Distributed Systems


Mr. Niket Subhash Doke (Research Scholar)

He is a student of BTech in Information Technology from Vishwakarma Institute of Technology. He has completed an internship at Chainworks Digital LLP as Associative Blockchain System Engineer. His interests include Blockchain, Machine Learning and Distributed Systems. He has published paper in the field of ML and Blockchain


Mrs. Aparna Mete-Sawant (Faculty)

She is working as Assistant Professor in Information Technology from Vishwakarma Institute of Technology. Her interests include Operating Systems,Distributed Systems and Web Technologies.

.