**New York university**

# Eventually Consistent Key-Value Store

Prepared by

**Sravanii Namburi (sn3109)**
**Shriya Jain (sj3409)**

December 13, 2021

## 1. Introduction

Distributed key-value storage systems are the foremost scalable storage systems within the modern storage systems. Despite their simple interface, distributed key-value storage system design is complex as they have to replicate data in many servers to attain higher availability and consistency in case of failure of a node or during network failures. This replication process may add substantial latency to storage operations. For achieving strong consistency, a client may read and write a majority of replicas. Since the majority of replicas overlap, every read ensures the newest write. In contrast, eventually consistent systems may read and write on non-overlapping quorum systems. Once a write is acknowledged, the new value is propagated to the remaining replicas asynchronously, thus, all replicas are eventually consistent unless a failure occurs. If the client fetches the data from replicas that haven't yet received the update, stale values are observed.

### 1.1 What's wrong with strong consistency?

Although many applications benefit from strong consistency, applications that are latency sensitive choose eventual consistency to achieve lower latency. Even though there are consistency anomalies due to reading of stale data, these anomalies are tolerable provided they are short-lived and very rare. This paper demonstrates designing and building an eventually consistent key-value store and analyse various hypotheses for quantifying consistency in such storage systems. This analysis is important for meaningful comparisons among different system configurations and workloads.

### 1.2 Design Considerations and Assumptions

- Non-overlapping quorums can lower latency by requiring fewer replicas to respond before replying to the client. Given N replicas and read and write quorum sizes R and W, partial quorums imply $R+W \leq N$ . In our key-value store, we are using partial quorums.
- We assume that there aren't any dropping of messages within the network.
- All the operations are considered atomic. Our datastore is currently limited to only GET and SET operations.

## 2. System Architecture

The architecture of a Key-Value store has to have solutions for scalability, failure detection, replica synchronization, and concurrency for better performance and durability. We'll start with designing a basic Key-value store then add failure detection and anti-entropy for replica synchronization functionality.

### 2.1 Sending Requests

The client selects the random server and sends requests to the selected server. For the SET request client sends a message {:set, key, value} to the replica. A {:get, key} message is sent to a random server during GET operation.

### 2.2 Replication

When the client request reaches the replica, it is in charge of the replication of the data item to all replicas asynchronously. As the response is sent to the client without completely replicating on all the servers but by satisfying quorum conditions, there may be chances for conflicting updates at different replicas for the same key.

Conflicts occur due to one of the following reasons:

1. When the two write requests for the same key goes to two different replicas concurrently, then there is a conflict for the values to be stored, when replication requests reach each other.
2. When the node is partitioned temporarily due to network delays, conflicts may occur when it re-joins the network again

If some sort of causal ordering is not maintained, the last write will win and useful data might be lost. To avoid this, data versioning using vector clocks is done.

**2.3 Data Versioning**

Our key-value store uses vector clocks in order to capture causality between different values of the same key. A vector clock is effectively a map of (node, counter) pairs. One vector clock is associated with every value of every key. One can determine whether two values of a key are concurrent or have a causal ordering, by observing their vector clocks.

Upon receiving a set() request for a key, the replica generates the vector clock for the new value and writes the new value and vector clock pair locally for the key. The replica then sends the value pair to all replicas. At each replica, if there are concurrent value-pairs (values with concurrent vector clocks) for the key, then all the values are stored or else the received value-pair is stored.

**2.4 Response to the client**

A node handling a client request collects responses from the members that it has replicated. When the number of responses for the particular replication request reaches R (for reading request) or W (for write request), then the response is sent to the client.

**Response for GET operation:** For a get() request, the replica requests all existing values for that key from all the servers and then waits for R responses before returning the result to the client. If the replica ends up gathering multiple values for the key, it removes the older versions and returns all the versions it deems to be causally unrelated.

**Response for SET operation**: When the replica gets a response for W replicas, a success message for that key is sent to the client.

The tables below show the algorithms for replication and response in replicas and state information of replicas.

**State Configuration For Replica**

view:[] // list of all members in the network
store: %{} // store key and value pair
clock: %{} // storing vector clock
reqnum: // for every request received to replica increase the reqnum
r: // Read Quorum
w: // write Quorum
merkle_version: 0 // version of merkle chain
merkle_hash: []
merkle_key_set: []
min_Merkle_timer:
max_Merkle_timer:
Merkle_timer:
Merkle_timeout
merkle_stat: %{success: 0, fail: 0}
gossip_timer // storing gossip timer reference
gossip_timeout // gossip timeout value
roundTrip_timeout: // storing roundtrip timer for gossip protocol direct ping timeout
roundTrip_timeout: // storing roundtrip timer reference
pr: // current gossip protocol term
subgroup_size: //size of indirect ping neighbours for gossip

replica(state, req_state, value_state)
req_state :%{} map containing reqnum as key and value as tuple of {count,client_id}
count is used to maintain quorum responses for that reqnum
value_state : %{} map containing reqnum as key and value as list of value-pairs
received during get operation.

**REPLICATION REQUEST**

key // key to be inserted or get from the store
value-pair // value and vector clock pair to be inserted for set operation and nil for get value
reqnum // count associated with the request
op //operation set or get

Receiver Implementation

1. If the operation is :get, return value-pair list for key if present or else []
2. If the operation is set,
   a. If the key is present in the store, compare all the list of value pairs with the current value pair using vector clocks in them and filter concurrent value pairs if present.
   b. update state clock by combining state. clock with vector clock present in the value-pair
   c. return :ok message

**REPLICATION RESPONSE**

key // key to be inserted or get from the store
value-pairs // value and vector clock pair to be inserted for set operation and nil for get value
reqnum // count associated with the request
op: //operation set or get

Receiver Implementation

1. If the operation is :get,
   a. After receiving a response, compare value-pairs received with the value-pairs present for the received key in the value_state map
   b. filter all the value-pairs that has vector clock before and store them back in the value_state map
   c. increase the count of the number of the responses for the given reqnum
   d. If the count reaches R,{value-list,key} is sent to client
   e. delete reqnum key in both req_state and value_state
2. If the operation is set,
   a. Increase the count of the number of the responses for the given reqnum
   b. If the count reaches W, {:ok, key} message is sent to the client
   c. delete reqnum key in req_state

**Rules for server receiving request from client:**

On receiving client get request {client_id,{:get,key}}:

1. increase reqnum
2. put {reqnum: {0,client_id}} key-value pair in the map req_state
3. put {reqnum: []} key-value pair in the map value_state
4. create replication request with (key, nil, reqnum, :get)
5. broadcast request to all replicas

On receiving client set request {client_id,{:set,key,value}}:

1. update the state vector clock
2. increase the reqnum
3. put {reqnum: {0, client_id}} key-value pair in the map req_state
4. create value-pair with value, clock
5. insert the value-pair in the store
6. create replication request with (key, value-pair, reqnum, :set)
7. broadcast request to all replicas

### 2.5 Membership and Failure Detection - Gossip Protocol

In our Key-Value store, if the client sends a set() or get() requests to one of the random nodes that have failed or are unreachable, then the client sees no response within the specific timeout. To avoid attempts to communicate with unreachable nodes and to maintain high availability, failure detection and membership update is required when nodes fail in the network. A membership protocol provides each process of the network with a view of other non-faulty processes in the group. The protocol ensures that

the view is updated with changes resulting from new members joining the group, or dropping out (either voluntarily or through a failure) and the clients can send requests to only available nodes, thereby reducing failed requests and increasing durability.

Decentralized failure detection protocols use a simple gossip-style protocol that enables each node in the system to learn about the arrival or failure of other nodes.

**The SWIM Approach**
(1) a Failure Detector Component:- detects the failure of nodes
(2) a Dissemination Component:- spread fail nodes status to remaining information

**SWIM Failure Detector:** After every gossip protocol timeout GT, a random member is selected from the process (say A) view (say B) and a ping message is sent to it. Awaits for a replying back from B. If this is not received within a prespecified time-out RT, then A indirectly probes B. A selects k members from its view at random and sends each a ping-req message. Each of these members in turn (those that are non-faulty), on receiving this message, pings process B and forwards the ack from B (if received) back to A. At the end of this gossip protocol period, process A checks if it has received any acks, directly from or indirectly through one of the members; if not, it declares as failed in its local membership list.

**Dissemination:** After detecting the failure of a node in the group, the detected process simply sends this information to the rest of the process in the view as a failed message. A process receiving this message deletes from its local View.

After Gossip Timeout at process A:

1. pr = pr + 1 //pr is the local term number
2. Select random replica B from view
    a. send a {:ping, pr} message to B
    b. Wait for the RT time units (round-trip time) for an {:ack, pr} message from B
3. If haven't received an {:ack, pr} message from B,
    a. select k members randomly from the view
    b. send each of them a {:pingreq, A, B, pr} message
    c. wait for an {:ack, pr} message until the end of protocol term
    d. If haven't received an {:ack, pr} message, declare B as failed

Receiver Implementation at A:

1. Reply with {ack, pr) to C, if receives {:ping, pr} from C
2. Reply with {:indirectack, B, C, pr) to D, if receives {:indirectping(B ,C, pr) message from D
3. If {:pingreq, C, B,pr} was received from C
    a. send {:indirectping, B, C,pr) message to B
    b. on receiving {: indirectack, B, C, pr} message from B, send {ack,pr} to C.

**2.6 Anti-entropy:**

There can be instances when replication of certain messages doesn't complete ever due to arbitrary delays and node failures. To synchronise divergent nodes Merkle Chains are used for synchronization. Merkle Chains are used to detect inconsistencies in data and reduce the number of messages to achieve synchronisation among key-value stores.

Merkle chain order maps to the order keys (sorted order of keys). Merkle chain is a list where each index corresponds to the hash of key and value concatenated with the hash of index + 1 (Fig 1.). The structure makes sure that if the head of two the Merkle chains of two nodes is the same then the key-value store is consistent with each.

| | | | |
|---|---|---|---|
| **H4 = H(** **H(**H(4)<>H(val)**)** + **H**3 + H2 + H1**)** | **H3= H(** **H(**H(3)<>H(val)**)** + H2 + H1**)** | **H2 = H(** **H(**H(2)<>H(val)**)** + H1**)** | **H1 = H(** H(1)<>H(val)**)** |
| **Key: 4** | **Key: 3** | **Key: 2** | **Key: 1** |

Fig 1.

Synchronization happens by each node selecting a random node after Merkle timeout and sending a request for synchronization. Synchronization takes place in the following manner:

| **MerkleSynchroRequest:** | **MerkleSynchroResponse:** |
|---|---|
| 1. version: int<br>2. merkle_chain: []<br>3. match_entries: %{} | 1. version: int<br>2. matched_entries: []<br>3. success: true/false |

**Implementation:**

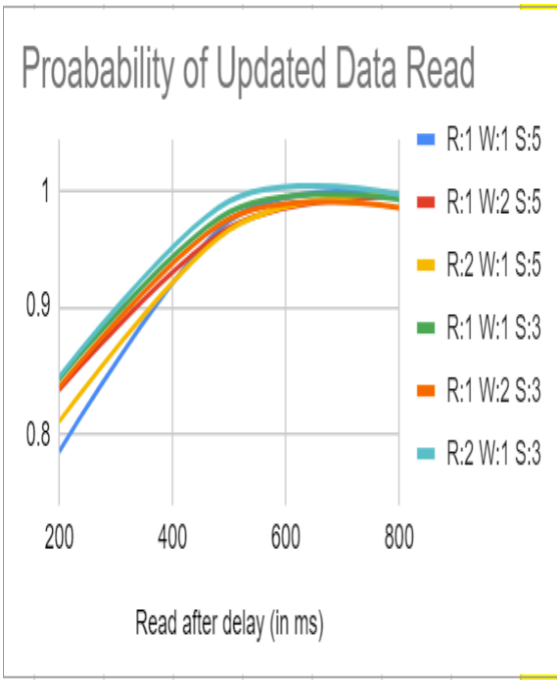
1. A sends B Merkle Request comprising of {current_merkle version, hash_list, %{}} after a timeout.
2. B compares its Merkle chain with A's Merkle chain and responds to A by sending the matched hashes (in order), the same version number as in request and success = false ({version, hash_list, false})
3. A checks the value of success in B's response. If success = true means A and B synchronization is successful. If success = false, then A compares its current merkle_version in its state machine with the version returned in the response.
    a. If the version matches: A send the request with a key-value map for all keys that didn't match, {current_merkle_version, hash_list, key_value_map}
    b. If the version doesn't match: The response from B is rejected and Step 1 is repeated.
4. B receives the request from A if the match_entries is empty then it sends back the matched hashes (Step 2). Else it sends {version, hash_list, true} back to A. B compares the received key values resolves the vector clocks and add the key values.

### 3. Results and Findings

We should be able to judge how our key-value store operates in practice. As eventually consistent data stores make no guarantees about the data they return, we can model their operation to know what consistency they provide using Probabilistically Bounded Staleness.

This models staleness according to two metrics: time and versions.

**1. Modelling using Time**:- The probability that the client reads a value, x seconds after its write request completes. This helps us to analyze: how eventual is eventual consistency? The graph below displays the probability of reading one of the last written data with varying delays between write and read.

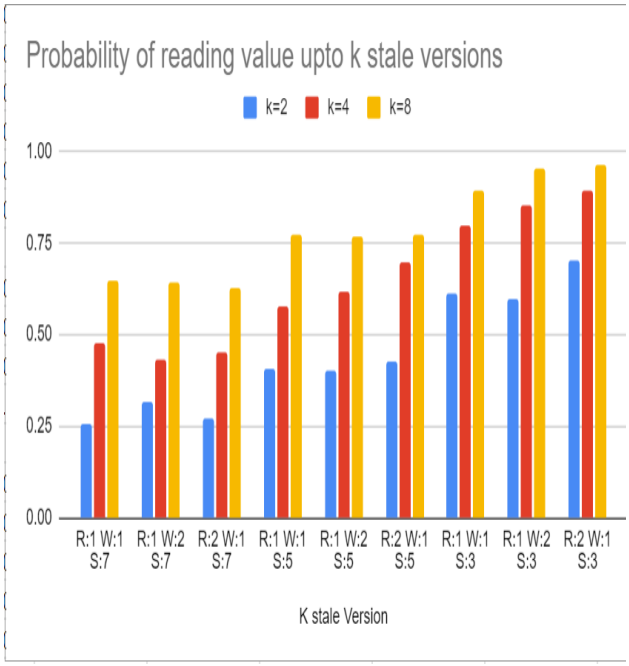



| Implementation of Reading Updated Value after Write Test. |
|---|
| count = 100;<br>Implemented using a delay of 50ms in the network<br>while count >0 do<br>1. The client selects the random replica and sends a write request for (key, value) = (“p”, 2)<br>2. wait for time t (in our implementation used 200ms,400ms,600ms,800ms)<br>3. The client again selects the random replica and sends the read request for the key =”p”<br>4. If the value returned is 2, increase prob_count.<br>5. count = count-1<br>return prob_count/100 |

From the above graph, we observe that for fixed R and W, the probability of reading the last written value increases with a decrease in N (replica size). For the fixed N, R> W has more probability of reading the updated value. As delay increases, the probability of reading updated value increases.

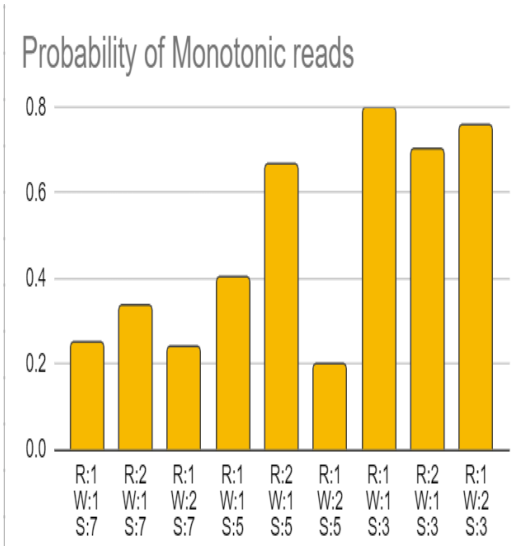
**2. Modelling using versions:-** The probability that client will read a value that is no more than k-versions older than the last written version. This helps us to understand: how consistent is eventual consistency.

From the below graph, we observe that for fixed R and W, the probability of reading value up to k stale versions increases with a decrease in N (replica size). For the fixed N, R> W has more probability of reading value up to k stale versions.




| Implementation of Reading Updated Value after Write Test. |
|---|
| count = 100;<br>Implemented using a delay of 50ms in the network<br>while count >0 do<br>1. The client selects the random replica and sends a write request for (key, value) = (“p”, 2)<br>2. wait for time t (in our implementation used 200ms,400ms,600ms,800ms)<br>3. The client again selects the random replica and sends the read request for the key =”p”<br>4. If the value returned is 2, increase prob_count.<br>5. count = count-1<br>return prob_count/100 |

3. **Monotonic Read Consistency**: If a client reads the value of the key p, any successive read requests on p will always return that same or a more recent value. Monotonic-read consistency ensures that if a client has seen a value of x at time t, it will never see an older version of x at a later time.




| Implementation of the probability of Monotonic Reads |
|---|
| Implemented using a delay of 20ms in the network<br>count = 10<br>while count >0<br>1. The client selects the random server and sends a written request for key “p” with a random value.<br>2. count = count-1<br>count = 10<br>while count > 0<br>1. the client selects the random server and sends a read request for key “p”<br>2. If the value read is at least as recent as the previous read, then increase prob_count;<br>3. count = count-1<br>return prob_count/10 |

From the above graph, we observe that for fixed R and W, the probability of monotonic increase with a decrease in N (replica size). For the fixed N, R> W has more probability of reading value up to k stale versions.

### 3.2 Merkle and Gossip protocol Analysis

**a**.) For write-intensive operations having a low Merkle Timeout results in high unsuccessful synchronization attempts. This defeats the purpose of Synchronization as the number of messages in the network increases considerably to achieve synchronization between two divergent nodes. Therefore the Merkle timeout for write-intensive key-value stores should be larger as compared to read intensive key-value stores.

**b.)** When there are node failures, the client request-response success rate is increased to 96 (using gossip protocol) from 78 (without gossip). This was tested on 100 client requests with server configuration N=5, R=1, W=1 and one node fails in the network.

## 4. Conclusion

The critical task in designing and building an eventually consistent datastore is to understand how consistency and latency are impacted by various configuration parameters, as well as the workload that is placed on the storage system. Only with proper insight into this system designers can make decisions regarding the configuration of the storage system or the choice of consistency model for a given application.

## 5. Future Work

**Extensions of the failure model:** In this implementation, we considered the failure of the single servers. It would be interesting to consider the failure of a link between two hosts, or a partition of the network and dropped messages. Further, the dissemination component of gossip protocol can be changed to infection style to reduce multicasting and adding a suspicion mechanism to failure detection will reduce false positives.

**Multi-key operations**: We have considered single-key operations and can be extended to perform multi-key transactions with considerable care in implementation.
We hope to further investigate these issues and their implication on our key-value store in future work.

**6. References**

1.DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., ... & Vogels, W. (2007). Dynamo: Amazon's highly available key-value store. ACM SIGOPS operating systems review, 41(6), 205-220.

2.Bailis, P., Venkataraman, S., Franklin, M. J., Hellerstein, J. M., & Stoica, I. (2014). Quantifying eventual consistency with PBS. The VLDB Journal, 23(2), 279-302.

3. Das, Abhinandan, Indranil Gupta, and Ashish Motivala. "Swim: Scalable weakly-consistent infection-style process group membership protocol." Proceedings International Conference on Dependable Systems and Networks. IEEE, 2002.

4. Gupta, Indranil, Tushar D. Chandra, and Germán S. Goldszmidt. "On scalable and efficient distributed failure detectors." Proceedings of the twentieth annual ACM symposium on Principles of distributed computing. 2001.

github code :- https://github.com/Shr2020/Distributed-System-Project