Key-Value Store Using Chain Replication

Ravi Pandey
ravip28@stanford.edu

Abstract- This paper is case-study of how chain replication protocol works and its implementation for a distributed key-value store. In theory chain replication protocol is easy to understand and reason about. In practice however, translating its formal definition and proof into effective code can be difficult because implementation must be correct and fast. Besides outlining the chain replication protocol, experiments to explore the performance of underlying key-value store is discussed.

1. Introduction

This paper is concerned about simple key-value storage service implementation that support following operations–

- Update (Key, Value) – atomically update key's value.
- Query (Key) - return key's value.

And service should be able to cope up with the high amount of request (high throughput HT, high availability HA) with strong consistency guarantees. I have used boltdb for the underlying key-value storage and we will focus on underlying replication technique in this paper.

Many systems sacrifice Strong Consistency to High Availability and throughput because its complex task to satisfy all three conditions.

Most popular approach to support these requirements is to use Primary/Backup (P/B) replication. P/B has one primary server and N backups. There are many variations of P/B setup like Primary does total ordering over all write requests, sending ACK as soon as one of the backups sent ACK, hinted handoff etc. A separate, stable process is needed to monitor cluster, and provide configuration. In case of failure of the primary node the process initiates new elections. Based on the requirement the process can be implemented as part of the replication algorithm or as an additional tool (like zookeeper). It is evident that performance of P/B approach is mostly bounded by the performance of Primary node and amount of backup nodes. More stringent the requirements, faster we will reach these restrictions.

Alternative approach is Chain Replication – In chain replication nodes are arranged in strictly ordered chain with two special nodes – HEAD (receives update request from client) and TAIL (end of the chain, provides the guarantee for consistency). Chain has following properties –

- Tolerates failure up to n-1 nodes, same as P/B.
- Write performance is around the same as P/B, however, has better throughput as load is shared between Head and Tail, whereas in P/B primary receives all requests.
- Reconfiguration is faster in case of Head failure, compared to other nodes which takes around same number of messages to be exchanged as in P/B.

2. Chain Replication

2.1. Overview

For chain replication to work following assumptions must be satisfied

- Servers are assumed to be fail-stop
    - Each server halts in response to a failure rather making erroneous state transitions

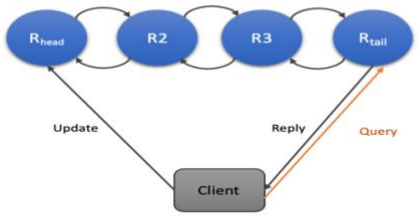
- A server's halted stated can be detected by the environment.

This is implemented in key-value service by simple alive (heart-beat) signals sent by each node. Node failure is detected when these signals are no longer emitted by it.

- Strong and reliable FIFO links between the nodes
  Current implementation uses TCP for order guarantees.

In chain replication the servers replicating the data (key value pairs) are linearly ordered to form a chain. The first server is called head, the last server is called tail, and request processing is implemented by the servers as follows–

- Reply Generation – The reply of every request (Update or Query) is sent by tail.
- Query Processing – Each query request is directed to the tail of the chain and processed there atomically using the replica of data store (boltdb database here) stored at the tail.
- Update Processing – Each update request is directed to the head of the chain. The request is handled there and forwarded until request is handled by tail.



A Chain

Strong consistency is thus followed because query and update requests are all processed serially at single server (the tail). Processing query request is relatively cheap as it involves only one server, but update request is propagated throughout the chain starting at head, thus around n-1 servers don't contribute to reply and only provide fault tolerance.

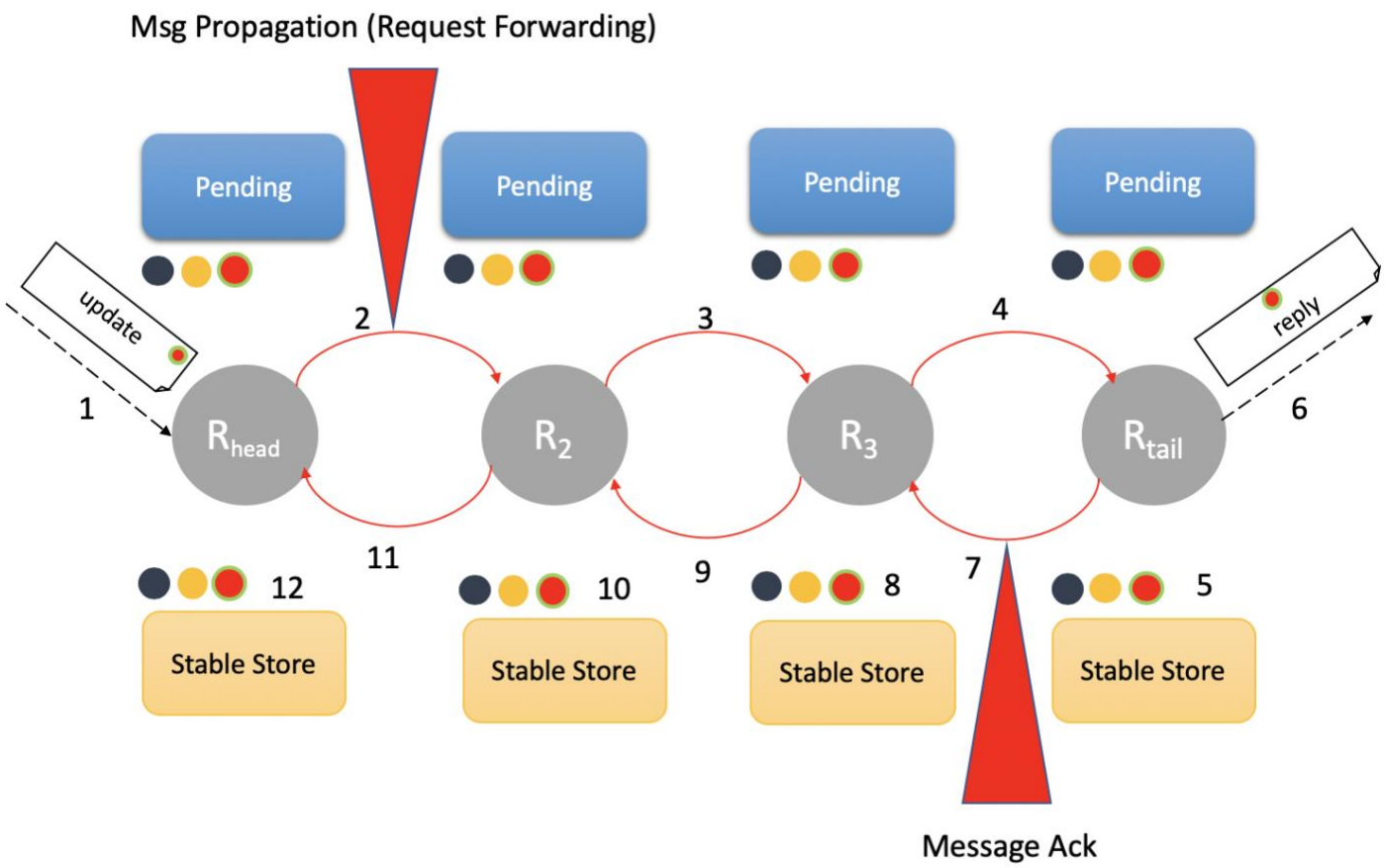
2.2. Details

Each replica keeps maintains following state (where 'i' denotes the i^{th} node in the chain) –

- Pending(i) – list of received requests by the node but not processed by the tail.
- Sent(i) – list of requests not processed by the tail but sent to successor of the i^{th} node.
- StableStore(i, key) – data store at node 'i' with current value of the 'key'

So following invariants are true at given point in time for a chain –

- StableStore(j, key) $\subseteq$ StableStore(i, key), $\forall$ j > i – an update record is not committed by node if it's not committed by all its successor node.
- Sent(i) $\subseteq$ Pending(i) – request can't be sent if not received.
- Pending(i+1) $\subseteq$ Pending(i), $\forall$I, request is always received by head and forwarded in chain in strict order.



Update Request Processing

2.3. Coping with Server Failures

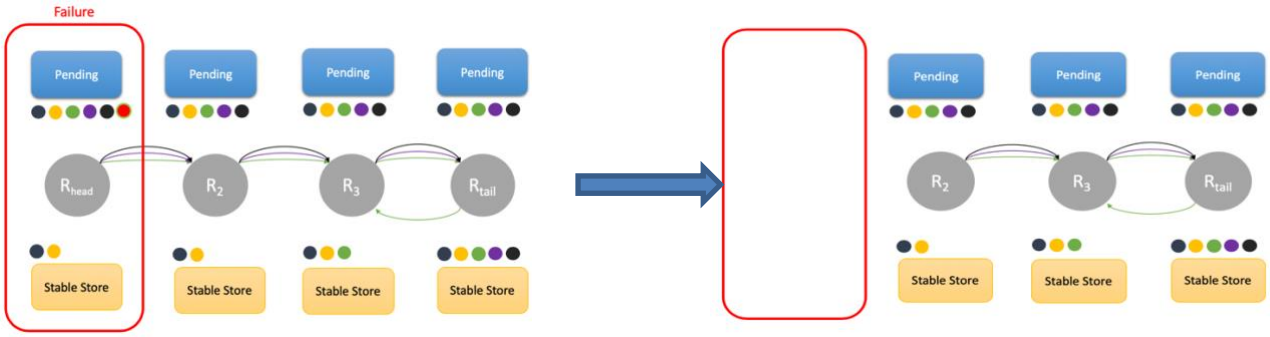
In response to detecting failure of server that is part of the chain (by fail-stop assumption, all failures are detected), the chain is reconfigured to eliminate the failed server. For this a separate service is created called **master,** as part of the implementation whose responsibilities are –

- detect that node has failed – every node is aware of the master and sends alive signals (RPC call) at regular intervals, master detects that node has failed, it didn't receive signals from the given node in particular checkup cycle (in this implementation checkup interval is 5s and alive/heartbeat signals are sent at 2sec interval)
- informs each server in the chain of its new predecessor or successor in the new chain obtained by deleting the failed server (this is done RPC call in current implementation).
- Updates the internal state about current head and tail so that it can inform clients about current config.

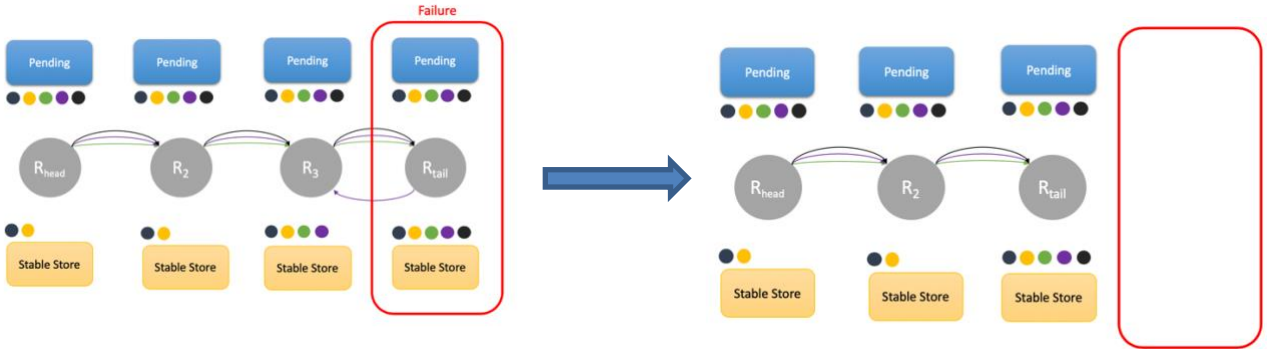
Implementation assumes that master never fails.

There can be following possible failure scenarios –

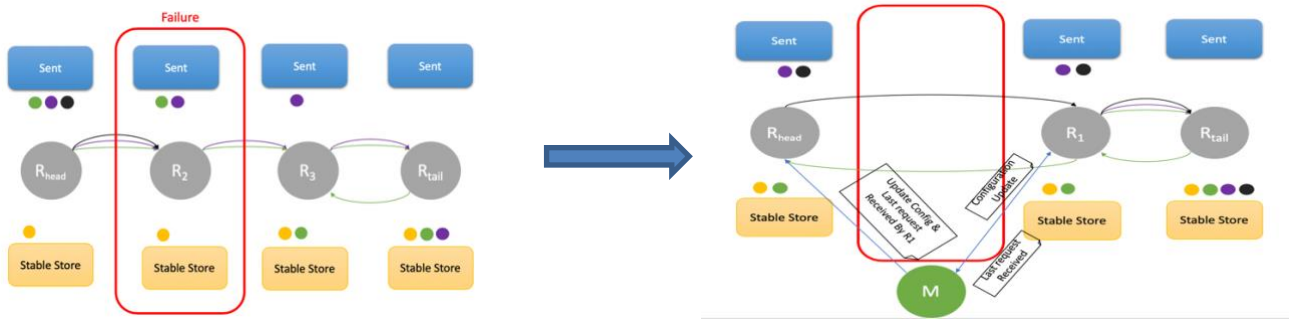
- **Failure of head** – Read requests are still served. There is delay of 1 message, master removes the head from the chain, makes its successor new head. There is possibility of message loss, given head received the update request and failed to forward it i.e. $Pending_{head}$



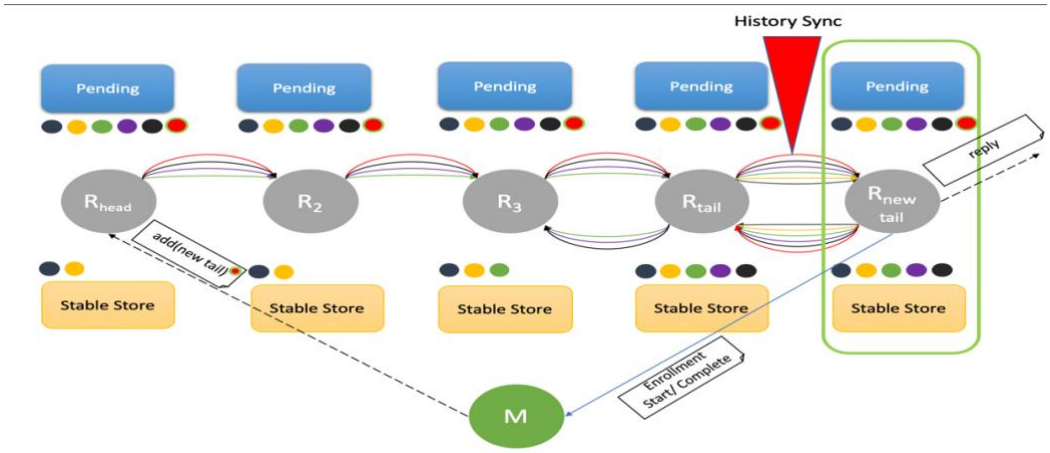
- **Tail Failure** – Delay of around 1 message for read and write requests, master removes the tail from the tail, inform its predecessor that it is the new tail. No update msg will be lost in this scenario. New tail immediately commits its pending requests and starts serving requests.



- **Middle Node Failure (i^{th} Node)**– Master notifies node i+1 about its new predecessor i-1, in reply it gets last request processed by i+1, it then informs i-1 about its successor and request processed by it, if any requests ($Sent_i$) are not received by i+1 and those requests are resent, thus no message is lost.

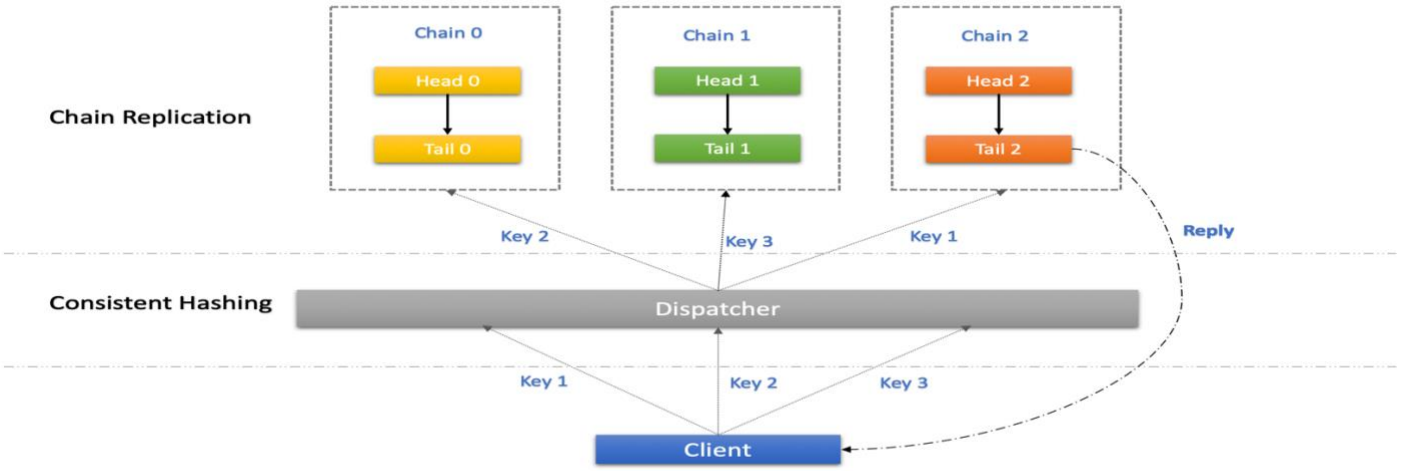


Extending the chain – Failed servers are removed from the chain. But shorter chains tolerate fewer failures, availability can be compromised if there are too many server failures in short time. Solution is to add new servers when chain gets short. In current implementation node is always added to the end (tail), as it is easy to reason about and handle failures. Chain extension is done by sending special request called add_{node} to the head of the chain by the master. When this request reaches the current tail, tail sync it's data-store with new prospect tail, and then master is informed, and the node becomes the new tail of the chain. Any failure is handled in similar fashion to that of update request.



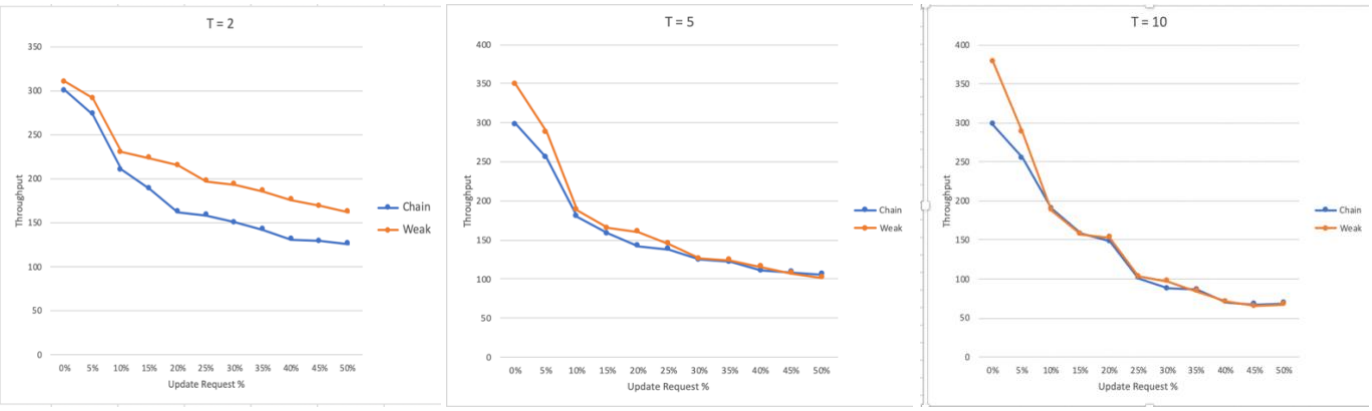
2.4. Multiple Chains for Data Partition

To support large scale services where data can't fit in single server, multiple chain configuration is supported as part of this implementation. For this a separate service called Dispatcher is deployed which is responsible for consistently mapping key to a given chain. Client proxy contacts dispatcher for update/query, dispatcher then forwards the request to appropriate chain, reply of any request is sent by the tail of the given chain directly.



3. Experiment And Observation

It is evident that update requests in chain-replication has to go through each server in the chain, so latency will add up because of the serial nature of request propagation, and only tail is used for queries, so not all servers are utilized. To find how throughput of chain replication for the current key-value service is impacted by read to write ratios. I had performed certain experiments to see how throughput is impacted if consistency guarantees are relaxed. Current implementation also supports weak chain configuration (query can be served by any node) which provides eventual consistency. For the experiment I had used 25 clients sending requests to chain in parallel with mix of different read to write ratios. The results are listed below (T = number of failures chain can tolerate)



Results that when are more than 15% update requests there is no difference in weak-chain and strong-chain variant and in some instances, one outperforms the other. Weak chain version has better throughput for query-heavy loads as query request is distributed over all the servers, but once the update requests increases strong chain out performs the weak chain in some instances as queries are not delayed at head awaiting completion of update requests, and there is more capacity at head as no query request is sent there.

Results for multi chain configuration is shown below for different Update Request %

It is evident from the results partition the requests among multiple chain clearly results in better throughput, where read only requests have higher throughput compared to higher percentage of update requests.

4. Conclusion

Chain replication supports high throughput for query and update requests, high availability with strong consistency. Key value service implemented using chain replication exhibited such properties as transient outages is no different from lost messages and doesn't introduces any new failure modes that client is not aware of. Also, as I learning I found that correct implementation of the protocol from formal definition presented in paper, is bit tricky as many implementation details are missing and not very obvious in paper and we have put conscious effort during implementation so that proofs and definitions are not violated.

5. References

I had only referred this paper Chain Replication for Supporting High Throughput and Availability (https://www.cs.cornell.edu/home/rvr/papers/OSDI04.pdf) along with lectures and notes from our distributed systems class for this implementation.