AOI: 10.100.234512.00…
13th international conference on Microelectronics, Circuits and Systems, Micro2026

Design and Implementation of Consistent Key-Value Store Based on Modified Chain Replication Protocol

(Distributed Key-Value Store with Chain Replication)

Adarsh Mittal[1], Ishan Kumar[2]
[1]University of Wisconsin Madison, USA
[2]Texas A&M University, USA
Corresponding author: adarshm9031@gmail.com

ABSTRACT

Replicated key-value stores sit at the core of many distributed applications, where they are expected to serve requests quickly, keep data safe, and keep working when individual machines fail. Building one is largely a question of how much consistency to trade away for availability and speed, and the answer gets harder once crashes and reconfiguration enter the picture. In this paper we describe a key-value store built around a modified chain replication protocol. It keeps data strongly consistent and durable, and stays available as long as a majority of the replicas are up. We cover the client-server protocol, how the system reacts to failures, and how state is stored on disk, and we measure the result under several workloads. Across different key and value sizes, client counts, and read-write mixes the system behaves predictably, and it stays correct when nodes go down.

Keywords: Distributed systems, Chain replication, Key-value store, Strong consistency, Fault tolerance, gRPC, Leader election, Replication chain, Durability, Availability.

I. INTRODUCTION

This paper discusses the implementation and evaluation of a durable, available, and consistent key-value storage service. The service provides a simple key-value store running as a set of replicated instances. The service is made durable and available, i.e., it will survive program and system crashes and continue to provide service when a majority of the instances are running.

The service is exposed to the client through a shared library that consists of five functions: init(), put(), get(), shutdown(), and die(). The init() function takes in a list of server names (as DNS or IP addresses) and a port number and initializes the connection between the client and those servers. The put() operation writes or updates a value for the specified key. The get() operation returns the value for the specified key. The die() function terminates a server, optionally notifying other servers in its partition depending on the clean option provided by the user. Finally, shutdown() closes the connections between the client and the servers.

The rest of the paper proceeds as follows. Section II surveys related work. Section III walks through the implementation, covering the client-server protocol, the gRPC layer, and the SQLite-backed storage. Section IV reports the tests we ran for correctness, availability, consistency, and durability, and Section V the performance numbers collected under uniform-random and hot-cold workloads. Section VI concludes.

II. LITERATURE SURVEY

Distributed key-value stores have been a long-standing topic of study because so many scalable cloud services are built on top of them. Dynamo was an early example: it leaned on replication and quorum techniques to stay available through failures, but gave up some consistency to do so [1]. Bigtable and HBase took a different route, building structured storage on top of distributed file systems and relying on sharding and replication for scale and fault tolerance [2][3].

How much consistency to offer has always been the difficult question in replicated storage. The CAP theorem made the underlying tension explicit — a system cannot have consistency, availability, and partition tolerance all at once — and a range of practical consistency models grew out of that observation [4]. At the strong end, primary-backup schemes and consensus protocols such as Paxos and Raft give linearizable behavior, though they pay for it in coordination overhead [5][6].

Coordination services like ZooKeeper and etcd build on consensus to hand distributed applications a set of strongly consistent primitives [7]. Spanner pushes the idea further still, pairing replication with tightly synchronized clocks so that transactions stay externally consistent even across the globe [8].

Our system is also a replicated key-value store, but its priorities are durability and consistency in the face of crashes. Unlike production systems, it deliberately keeps things simple and its semantics easy to follow, which makes it a good fit for teaching and experimentation without straying from the core ideas of distributed systems.

A. Motivation and Contribution

Key-value stores have been studied at length, yet most of them either relax their consistency guarantees or take on fairly involved coordination machinery to keep them. We wanted to show that a system can be strongly consistent, durable, and available without being hard to understand. Our main contribution is a modified chain replication protocol in which the head node answers both reads and

writes; this keeps the client side simple while still preserving consistency. Beyond that, we handle the messier parts in full — crash failures, leader-driven reconfiguration, and bringing recovered replicas back in — and we validate the whole thing experimentally rather than leaving it on paper.

B. Challenges and Research Gap

Most existing stores commit to either strong consistency or high availability, and the trade-offs that follow are hard to untangle in a classroom or lab setting. Production systems make this worse from a learning standpoint: features like multi-version concurrency control, synchronized clocks, and sharding across geographically separated replicas tend to bury the basic ideas of replication and consistency under layers of engineering. What is missing is a system that shows off strong consistency clearly and is still small enough to validate piece by piece.

C. System Model

The system is a fixed set of storage servers arranged as a replication chain, plus a single logically centralized leader that tracks membership and watches for failures. We assume a fail-stop model: a crashed node simply stops rather than behaving arbitrarily, and it rejoins only through an explicit recovery procedure. Clients reach the servers through a shared library using synchronous RPCs, and every replica keeps its state on local disk so that data survives a crash.

III. IMPLEMENTATION

A. Chain Replication

We use a modified version of chain replication in the design. Figure 1 shows the implementation of chain replication. In our design, the head node services both PUT and GET requests, unlike the original chain-replication design where the head processes PUTs and the tail processes GETs. Since the client library does not support asynchronous calls, we could not send a request to one node (the head) and receive the reply from another (the tail). To simplify the design, we send both PUT and GET requests to the head node. Chain replication achieves coordination between fail-stop storage servers in the cluster and provides high throughput and availability without sacrificing strong consistency guarantees.

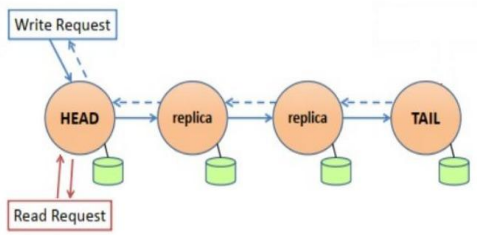

Fig. 1. Chain Replication Protocol

B. Client-Server Protocol

The system uses the gRPC protocol for communication between server and client. We define the KeyValueStore service on the server side, specifying the RPC methods that can be called remotely with their parameters and return types. The server runs a gRPC server and handles client calls. The client has a stub that provides the same methods as the server. By default, gRPC uses protocol buffers — a mechanism to serialize structured data — where each message is a small logical record of information containing fields.

In addition to the head and tail nodes, we also have a leader node which stores information about the current head of the chain. Prior to any request, the client library first contacts the leader to obtain the address of the current head. The client makes a GetHead request to the leader, which returns the address of the head node.

C. Get Request

The client makes a Get request to the head node. For the Get request, the head node does not need to communicate with the rest of the servers; it fulfills the request locally. If the requested key is found, the call returns the corresponding value with status code 0. If the key does not exist, the call returns status code 1. On a network error, the client library returns -1 indicating a failure.

```
message GetRequest {
  string key = 1; //Request to get the key
}

message GetResponse {
  int32 status = 1; //Response status
  string value = 2; //Response value for requested key
}
```

Fig. 2. gRPC messages for GET

D. Put Request

The Put request must be replicated to all replicas in the chain. The client makes the Put request to the head node, providing the key and value. The head node makes a PutServer request to the next node in the chain, also carrying the key and value. The next node again makes a PutServer request to its successor. This process continues recursively until the tail node, which writes the key and value to disk and returns the PutServer response with status 0. Each preceding node, upon receiving the success response, writes the key-value pair to its own disk and returns. This chain of returns continues to the head, which finally persists the key-value pair and returns to the client with the old value (if any) and status code 0 (if there was an old value) or 1 (no old value). If the PutServer request made by the head fails, the head returns status -2 to the client, allowing the client library to retry. The old value (if any) is reported based on the state of the head node only.

```
message PutRequest {
     //Put operation with key and value
     string key = 1; string value = 2;
}

message PutResponse {
     //Response status and old value
     int32 status = 1; string old_value = 2;
}

message PutServerRequest {
     //Put request to successor node
     string key = 1; string value = 2;
}

message PutServerResponse {
     //Put response from the sucessor node
     string oldValue = 1;  int32 status = 3;
}
```

Fig. 3. gRPC messages for PUT and PUT_SERVER

E. Die Request

In addition to GET and PUT, the service supports a DIE request. The DieRequest message includes a clean bit: if clean is set to 1, the server gracefully terminates by notifying other servers; if clean is set to 0, the server terminates without notifying others.

F. Handling Node Failures

Head Server Failure: The failure of the head node (or any other node) is detected by the leader. If the head fails, the leader updates its records and promotes the failed head's successor to become the new head. Henceforth, the leader returns the address of the new head for all subsequent GetHead requests, directing all Get/Put traffic to the new head. If the old head later restarts, it notifies the leader and is added to the end of the chain.

Intermediate Server Failure: The leader pings every server with periodic heartbeats. When one stops replying, the leader treats it as failed and reconfigures the chain around it, issuing a SetSuccessor RPC to the dead node's predecessor that tells it who its new successor is.

Tail Server Failure: If the tail stops answering heartbeats, the leader marks it failed, drops it from the chain, and promotes its predecessor to tail with a SetTail RPC.

G. Server Revival

When a failed server comes back up, it is added to the end of the chain and appointed as the new tail. The process is as follows: (1) the recovering server makes an AddChain RPC to the leader; (2) the leader makes a StartTransfer RPC to the current tail, directing it to transfer its data to the new server; (3) the tail makes a DataTransfer RPC to the new server, copying its database contents; (4) on success, StartTransfer returns SUCCESS to the leader; (5) the leader makes a SetTail RPC to the new server, making it the new tail; (6) the leader makes a RemoveTail RPC to the previous tail, after which the previous tail continues as an intermediate server. During this process, PUT requests are not serviced; however, GET requests continue to be accepted.

H. Leader Election

The system uses the Kazoo Python library for leader election. The algorithm uses a distributed coordination service to manage a large set of hosts. This implies that the master is a single logical process that never fails. Other commonly-followed approaches for leader election include Paxos and Raft, which coordinate replicas so that they behave like a single non-failing process.

I. Back-end Storage

The system uses SQLite3 as the persistent storage backend. SQLite3 implements transactions that are atomic, consistent, isolated, and durable. We use SQLite3 in its default full-synchronous mode, ensuring that all content is safely written to disk and that an operating system crash or power failure will not corrupt the database. Each server has a separate SQLite3 instance; together they act as multiple replicas of the database, and the chain protocol coordinates them to ensure consistency.

IV. VALIDATION

A. Correctness Tests

Single Client: To check correctness in the absence of failures, we start five server instances and initialize a single client. The client performs PUT operations with varying key and value sizes and then GETs on the keys written, comparing the values. We also generate a fixed-length set of keys and values, PUT them, and GET them back, checking the return codes (0 for the first PUT, 1 for an updated key, and so on). We also perform a GET on a key that was never written, expecting return value 1. This sanity check confirms that init, put, and get work as expected and return correct values for inserting new keys and updating existing keys.

Single Client with Server Failure: To confirm that the service recovers correctly and no data is lost, we start five server instances, write 1000 key-value pairs, then kill the head node. The client reads the same keys and we compare the values. The experiment is repeated for killing an intermediate node and the tail node.

Multiple Clients, Multiple Servers: We start five server instances and fork the client to create two processes (client 1 and client 2). Client 1 PUTs a key-value pair and client 2 reads it, repeated 1000 times for client 1 and 10000 for client 2, with two semaphores enforcing ordering.

Parallel Put and Get: With a single server, one thread writes a new value to an existing key and another thread polls GETs. When the new value is observed, the test kills and restarts the head without waiting for the previous PUT to return, then re-reads the key to check the value. The experiment is repeated for intermediate and tail failures. The GET request should not return a new value until the corresponding PUT succeeds.

B. Availability and Consistency Tests

We performed tail-, head-, and intermediate-node failure tests. In each test, five server instances are started and 1000 key-value pairs are inserted. The target node is then killed, 1000 GETs are issued on the same keys, and returned values are compared against the originals. The cycle of inserting 1000 pairs and killing one server is

repeated until only a single server instance remains; PUTs and GETs are again performed and verified. The implementation provides the expected results even when only a single server is running, so the system is highly available.

C. Durability Tests

To validate durability we start five server instances, PUT 1000 key-value pairs, and after all PUTs return SUCCESS we call die() on all servers. We then restart just one server, connect to it, and perform 1000 GETs to check the values. Since replication succeeded before the crash, we are able to read back the values, confirming that the system is durable.

V. PERFORMANCE EVALUATION

We measured performance by varying key-value sizes, the number of key-value entries (database size), the number of clients, and the ratio of put-to-get operations in the workload. Tests use randomly-generated alphanumeric keys and values (default 128 B and 1024 B respectively). We performed 50,000 operations with both uniform-random distribution and a hot-cold distribution (90% of the requests targeting 10% of the keys). Tests were run on the CSL machine royal-04 (4 cores).

A. Different Key-Value Sizes

Figure 4 captures the latency and throughput for sizes (2,16), (4,64), (16,256), and (64,1024). Latency and throughput remain almost identical across sizes, because every PUT persists changes to disk and a read/write touches one disk page per operation.

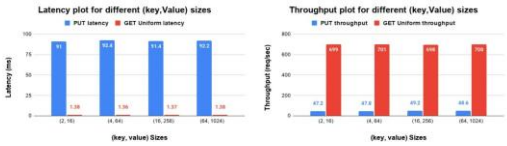

Fig. 4. Varying (key, value) sizes: latency and throughput

B. Increasing the Number of Clients

We varied the number of clients from 2 to 32 and measured throughput and latency. Figure 5 shows a small increase in latency and a small decrease in throughput with the number of clients. This is due to increased contention when clients establish the gRPC channel simultaneously.

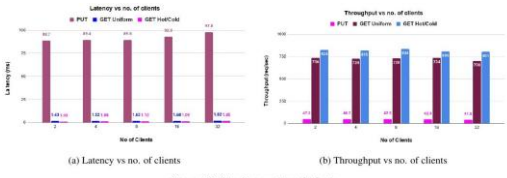

Fig. 5. Varying number of clients: latency and throughput

C. Number of Key-Value Entries

We varied the number of stored key-value entries from 500 to 4000. For each datastore size we performed 50,000 GETs under uniform and hot-cold distributions. Figures 6 and 7 show that PUT is an order of magnitude slower than GET, because every PUT propagates through the chain. Read latency is essentially independent of the read distribution; throughput is slightly higher (about 100 reqs/sec more) under hot-cold. Overall, write throughput averages around 50 reqs/sec and read throughput averages around 800 reqs/sec.

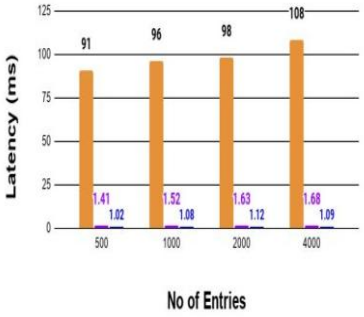

Fig. 6. Latency vs. number of entries in KVS

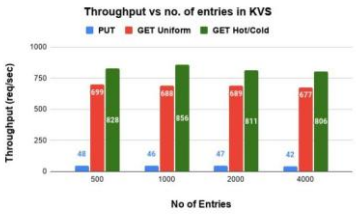

Fig. 7. Throughput vs. number of entries in KVS

D. Get-Put Mixes

We tried get-put ratios of 50:50 (balanced), 75:25 (read-mostly), and 90:10 (read-heavy), each running 50,000 mixed operations under uniform and hot-cold distributions. Figures 8 and 9 capture the uniform distribution; Figures 10 and 11 the hot-cold distribution. PUTs remain an order of magnitude slower than GETs. As the PUT fraction decreases, throughput rises and latency falls, because PUTs are costly: changes propagate through the entire chain before the success code returns to the client.

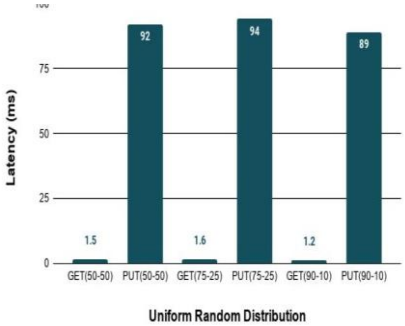

Fig. 8. Latency for uniform random distribution

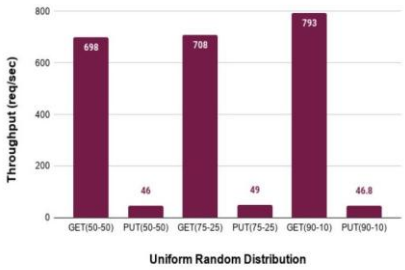

Fig. 9. Throughput for uniform random distribution

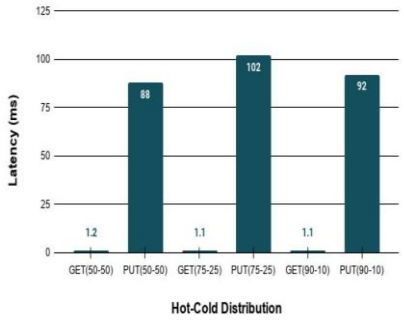

Fig. 10. Latency for hot-cold distribution

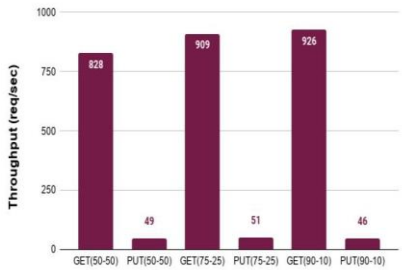

Fig. 11. Throughput for hot-cold distribution

E. Assumptions and Limitations

We assumed servers are fail-stop and that, when restarted after a failure, they contact the leader node. The design has a few limitations. First, PUT latency is high because the request must be propagated through the entire chain. Our chain replication has higher GET latency than the original chain replication because GETs are also served from the head node. Second, the implementation suffers from a load-balancing problem: the head node handles the majority of the system load since all servers are part of a single chain. Having multiple chains with different nodes serving as the head for each chain can address this, and we are working on this extension. Additionally, while adding a recovering server to the chain we do not service PUT requests; GET requests, however, are accepted.

VI. CONCLUSION

We have described and evaluated a consistent, durable key-value store built on a modified chain replication protocol. The takeaway is that strong consistency and fault tolerance do not require a complicated design: ours stays easy to reason about and to test, and the measurements bear out predictable behavior across the workloads and failure cases we tried. For now everything runs on a single chain. The natural next steps are to run several chains at once and to add an asynchronous client interface, both of which should help with scalability and with spreading load more evenly.

ACKNOWLEDGEMENT

We are grateful to the University of Wisconsin-Madison and Texas A&M University for access to the research infrastructure used in this work, and to the peers who offered feedback along the way.

REFERENCES

[1] G. DeCandia et al., "Dynamo: Amazon's Highly Available Key-value Store," in Proc. ACM SOSP, 2007, pp. 205–220.

[2] F. Chang et al., "Bigtable: A Distributed Storage System for Structured Data," in Proc. USENIX OSDI, 2006, pp. 205–218.

[3] L. George, HBase: The Definitive Guide, 1st ed. Sebastopol, CA, USA: O'Reilly Media, 2011, ISBN: 978-1-449-39610-7.

[4] E. Brewer, "Towards Robust Distributed Systems," Keynote, ACM PODC, 2000.

[5] L. Lamport, "Paxos Made Simple," ACM SIGACT News, vol. 32, no. 4, pp. 18–25, Dec. 2001.

[6] D. Ongaro and J. Ousterhout, "In Search of an Understandable Consensus Algorithm," in Proc. USENIX ATC, 2014, pp. 305–319.

[7] P. Hunt et al., "ZooKeeper: Wait-free Coordination for Internet-scale Systems," in Proc. USENIX ATC, 2010, pp. 145–158.

[8] J. C. Corbett et al., "Spanner: Google's Globally Distributed Database," ACM Trans. Comput. Syst., vol. 31, no. 3, pp. 8:1–8:22, Aug. 2013, DOI: 10.1145/2491245.