MochiDB : A Byzantine Fault Tolerant Datastore

Tigran Tsaturyan
Stanford University

Saravanan Dhakshinamurthy
Citrix Systems

Abstract

In this paper we would like to present MochiDB - a consistent, high volume, distributed datastore which is Byzantine fault tolerant. MochiDB supports native transactions and uses BFT quorum protocol with random write seeds that requires only two round trips for writes and one for read which gives it low latency over WAN deployments. This paper puts focus on engineering solutions that minimize the cost of contention resolution, sharding, dynamic configuration changes, garbage collection and others.

1 Introduction

When designing MochiDB we were pursuing the use case of having consistent key-value datastore located within data centers around the globe supporting large volumes of data and transactions. We builtin Byzantine fault tolerance (BFT) support so that the system can remain consistent even if some nodes are malicious or have failed arbitrarily. Our concrete application was a infrastructure manager which centrally contains configurations for servers, VMs, docker images, site certificates, passwords, etc - i.e. everything that is read more often that is updated. Such information need to be consistent because small divergence can cause misconfigurations of the infrastructure and failures. We provide transaction support as very often multiple items are updated together.

2 System Overview

MochiDB consists of clients and servers that contain the distributed database where data is stored. We represent data as a key-value store, where to some string 'K' is assigned some string value 'V'. We do not put any limit on key and value length, but our expectations are that keys are less than 80 characters long and data can be up to dozen Mb. MochiDB works only on string key and string values. Every read and write for an object has to be encapsulated as a transaction. Clients execute transactions across servers to access data for read and write. Each transaction consists of a list of operations. Each operation can be READ, WRITE or DELETE. Table 1 on page 1 describes their meaning.

Table 1: MochiDB client operations

Operation	Input arguments	Result	Meaning
READ	Key	Current value	Read data mapped to key
WRITE	Key, New value	Old value	Write new data to key
DELETE	Key	Old value	Delete data for key

A Transaction in MochiDB is guaranteed to be executed atomically. However, the transaction cannot be rolled back and the effects can be undone only via a separate transaction. Clients nudge slow servers to resync objects that were found to be stale during a transaction.

Communication between participants (clients and servers) occurs via messages. MochiDB employs key authentication to guarantee that messages originated from the stated senders. The entire communication channel used for messages, data can be encrypted using TLS at the transport layer.

MochiDB is built to tolerate byzantine failures. The relationship between number of robust servers and number of faulty servers (or faulty replicas) is described using the following equation: $N = 3 * f + 1$ where N is total number of servers required (*without any sharding - see section later) and f is the presumed number of faulty servers.

Due to the nature of BFT protocol, we have to store relatively large metadata for any object in our datastore. To solve those issues, we introduced sharding - the possibility to split data across different machines. We present sharding more in depth in the subsequent sections.

MochiDB supports dynamic configuration changes. Clients with special privileges can add and remove servers without shutting down the system. But during

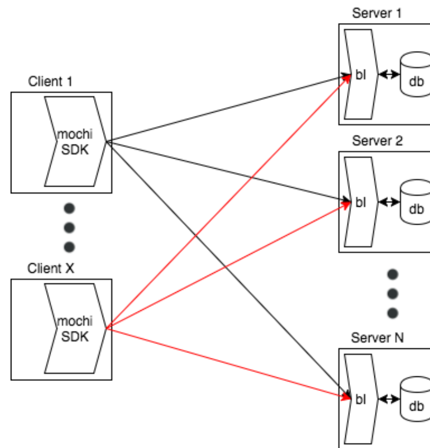
transition phase, read and write operations are put on hold.

2.1 Design Assumptions

When building MochiDB we took several assumptions which helped us to simplify the system and optimize relevant components. Since MochiDB is deployed over WAN in different parts of the world, communication between nodes is physically limited by underlying network. For example, ping time between Tokyo and Barcelona is around the magnitude of 300 ms and between Paris and Los-Angeles 150 ms [3]. Those times are multiple order of magnitude faster than CPU execution time. That means that optimizing for less number of message round-trips is better than optimizing for processing time. Hence, using public key cryptography as well as cryptographically secured hash functions do not dramatically bog down the overall cluster's throughput.

Figure 1: MochiDB System Diagram

Each of the clients uses MochiSDK to execute Mochi protocol. Each of the servers contains of business logic (BL) which is responsible for running protocol and DB (datastore) which contains objects and metadata.



3 Architecture

MochiDB is built to be an an asynchronous system "where nodes are connected by a network that may fail to deliver messages, delay them, duplicate them, corrupt them, or deliver them out of order, and there are no known bounds on message delays or on the time to execute operations." [2] MochiDB was largely influenced by HQ Replication [2]. Traditional agreement-based BFT protocols such as PBFT [1] require too many message exchanges during operations between servers, for every request from the client. Diagram 2a and 2b visualizes the message overhead. Such protocol performs well when servers are located close to each other and client is far. But in case of MochiDB, both servers and clients are located far from each other.

Our system consists of clients $C = \{C_1, C_2, ..., C_X\}$. And servers $S = \{S_1, S_2, ..., S_N\}$. Clients are servers are identified by random ID which is unique for each client and server. Clients act as coordinators for transactions. Servers store data inside internal DB (datastore) and reply to the clients by defined protocol. Sometimes servers can initiate communication and talk to other servers, for example to synchronize missing data. Diagram 1 visualizes our architecture.

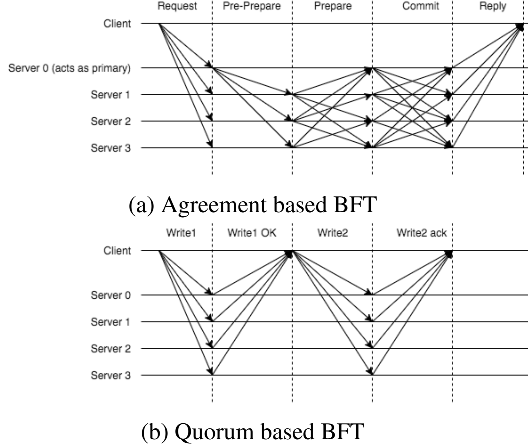
In the best case scenario, we require only one round-trip for read transaction and two for write transaction. Read transactions do not modify state of the object. Write transaction consisting of only write operations does modify the state of the object. Write operations can be overriding value or deleting object. MochiDB puts constraint that we do not allow mix of read and write operations within the same transaction. That was done mostly for simplicity, but we believe that it is not hard to design such support in future.

The server contains internal database (DB, datastore) which stores dynamic part of configuration, objects and metadata. DB is structured as keys mapped to StoreValueObjectContainers (SVOCs) and keys themselves represent object key. Internally SVOC stores value and metadata associated with object. For example, if MochiDB stores only two objects $O_1 = \{"Washington" : "Olympia"\}$ and $O_2 = \{"California" : "Sacramento"\}$ which maps states to their capitals, then DB will contain two keys - "Washington" and "California" . Those keys will be mapped to some $\{SVOC_1\} and \{SVOC_2\}$ respectively with values "Olympia" and "Sacramento" and metadata inside.

3.0.1 SVOC and Object Metadata

StoreValueObjectContainer contains the following data inside it:

- *value* - string value associated with that key. Can be null.
- *key* - string key for faster lookups
- *valueAvailable* - boolean property which indicates whether value is available. Non initialized or deleted keys will have valueAvailable set to false
- $\langle long, Write1MultiGrants \rangle$. - Maps $Epoch$ timestamp to collection of Write1Multigrants given on that $Epoch$(see section 4.2)
- *currentC* - current write certificate for that object - i.e. signed collection of grants by different servers with timestamp



(a) Agreement based BFT

(b) Quorum based BFT

Figure 2: (a) Agreement based BFT produces a lot of messages between the servers as well as require extra communication back to the client. Compared to (a) which requires less message and also less communications

3.1 Trust and Permissions

To confirm that message was truly send by some party, MochiDB uses signatures. As part of cluster config, a master certificate will be used as trusted certificate authority. Such master certificate is fixed for the life of the cluster and must be carefully guarded. The Master certificate is used to sign certificates for every client and server in the MochiDB cluster. MochiDB does not allow blacklisting server certificates, but it allows configuration change to remove faulty servers from the cluster. MochiDB allows clients with following levels of permissions : READ (read objects), WRITE (read and) and ADMIN (read, write objects and modify configuration).

4 Protocol Processing

In the core of MochiDB lies quorum based BFT protocol. Most of the actions (read, write, epoch progressing, GC, etc.) are done in one or two phase approach. During phase one, some member (client or server) initiates action and send it to all $3f+1$ nodes (Note: shading modifies that equation and is described separately. For the future descriptions assume that no sharding in present, unless specifically stated so). Then, the initiator waits for $2f+1$ (quorum size) matching responses for write and $f+1$ for reads. If server processing is required, then similar phase 2 starts: responses received during phase 1 are send back to the server along with some action. If some initial phase fails, initiator can suggest old servers to bring themselves upto speed and retry operation or message, which are explained in detail in the subsequent sections.

4.1 Reads

Client initiates read transaction and send readToServer message $\langle ReadToServer,\ transaction,\ nonce \rangle$ to all servers, where nonce is secure random number of uniquely identify request and avoid replay. Each of the servers reply back with readAns $\langle ReadAns,\ transactionResult,\ nonce \rangle$, where transactionResult contains list of OperationResult - one per each read operation. OperationResult has the format of $result,\ currentC, existed$, where $result$ - string result of that operation, $existed$ boolean indicator whether such value existed and $currentC$ is the current write certificate for that object. Client waits for $f+1$ matching responses and if found, return $transactionResult$. If matches were not found, that indicates that some servers are not yet up to date. After a small wait time, the client retries the READ. When client consistently sees grants with older TS, epoch they can initiate resync to bring slow mochiDB servers upto speed with other replicas. There is a risk that frequently updated keys will starve reads as there will be no quorum on the latest data.

4.2 Writes

Client initiate write transaction and send write1ToServer message $\langle Write1ToServer,\ transaction,\ txnHash,\ subEpoch \rangle$ to all servers, where $transaction$ contains just the keys as values are unnecessary in the write1 phase, $txnHash$ is the hash value calculated for the entire transaction including value (excluding value in case of delete) that will be stored in server; $subEpoch$ is the random seed, a number within 0-1000 range alluded earlier in this paper. As the value payload will be sent later, $txnHash$ is invaluable for servers to catch malicious clients who could flip the payload after aquiring a Write1Grant, by including it in the Write1Grant. Upon receiving write1 message, each server checks whether for each object there is already grant for timestamp $NextEpoch + subEpoch$, where $NextEpoch$ is the next timestamp epoch for that object and $subEpoch$ is the random number received from the client. MochiDB uses unsigned long format for timestamps. Each timestmap consist of epoch and numbering within epoch. We allocate 0-1000 for numbering within each epoch and the rest for epoch. For example, TS=4342 contains of epoch 4 (or equivalently, 4xxxx) and 342 as number within epoch. If timestamp is not granted to any other client, server creates write grant in form $\langle objectId, timestamp, configstamp, txnHash \rangle$. If timestamp was granted, mochiDB checks whether the $Write1ToServer$ was a retry because the previously

issued $Write1OkFromServer$ was lost by the network and responds back with the stored grant information, else the server denies issuing grant due to the client and responds with $\langle$ $Write1RefusedFromServer$, $currentC\rangle$. Note that grant is permission for a client to perform write operation on an object at specified future timestamp. Grants are not revocable - once granted, they can be executed at any point later on. Grants for all objects within transaction are unified under multiGrant and stored in the server so that the Server does not issue Write1 multigrants to other clients and also to service client retries within an epoch when $Write1OkFromServer$ is lost by the network. After stably storing multiGrant, server sends back $Write1Ok$ message $\langle$ $Write1OkFromServer$, $multiGrant$, $currentC\rangle$ where $currentC$ is the object's current certificate.

The client waits for $2f+1$ matching responses and construct $writeC$ out of received Write1 multiGrants and send $\langle Write2ToServer$, $writeC$, $transaction\rangle$ message to all servers. Servers processing Write2 message will verify whether the $writeC$ contains $2f+1$ grants with consistent timestamps, calculate transaction Hash based on the complete $transaction$ and verify it with $txnHash$ present on Write1 multiGrant; if all validation criteria are met, the Servers commit changes, save provided $writeC$ as $currentC$ for each object and delete the stably stored, corresponding write1Grant issued earlier in Write1 phase; the objects themselves might move to a NewEpoch if necesssary at the end of Write2 phase, more on that in the next section. If the Server finds a discrepancy in txnHash or writeC, the server rejects Write2 operation, responds with a blank $write2Ans$ signifying abort and allows for the client to retry. In the former case, each server sends $Write2Ans$ message which describes current state of objects involved in the transaction. The clients can conclude the transaction is committed when it receives $2f+1$ matching $Write2Ans$

4.2.1 Write contention

MochiDB server does not freeze read, write requests to perform contention resolution unlike other BFT schemes which perform conflict resolution via agreement that involve message overhead[2]. Neither does it assign dedicated primary server to resolve conflicts like PBFT[1]. The random seed passed as $subEpoch$ minimizes the probability of grant timestamp collisions. If two or more competing transactions do collide by having same $subEpoch$ value on overlapping objects, servers will nudge clients to retry with a $Write1RefusedFromServer$ message. Note that clients require $2f+1$ $Write1OkFromServer$ messages with consistent Write1 Multigrants to proceed to the next stage, when they fail to get the majority they have to perform retries with a new random $subEpoch$. On a key that is frequently written, it is possible for clients to receive $Write1OkFromServer$ from majority of servers but with multigrants with timestamps that don't match, even in such scenarios, the client has to wait for a short time and retry again to get a consistent majority grant. Each object in SVOC contains $currentC$ with timestamp inside it. This timestamp consists of $Epoch$ and numbering within the epoch - $subEpoch$. The Object might move from one epoch to another when Write2 message is processed and $currentC$ gets overwritten. Note that subsequent Write1 multiGrants are assigned from next epoch relative to the epoch of the $currentC$. Intuitively, epoch change acts as a synchronization point between transactions - if two transactions run at the same time, objects they modify should be from the same epoch. If one transaction runs after another is finished, object should belong to different epochs. Within each transaction different objects might be from different epochs, but subEpoch should be exactly the same. Diagram 3 visualizes epoch timeline.

When server executes Write2 transactions which are old, current and advanced relative to currentC's epoch, the following rules take affect: If epoch of transaction T is less than current epoch of an object, the object will not get modified. If epoch of T is the current epoch, but subEpoch is less than currentC's subEpoch, the object will not get modified. In both cases, the server replies success with Write2Ans along with most recent object value. If epoch of T is the current epoch, but subEpoch is large than currentC's subEpoch or the epoch of T supersedes currentC's epoch - update object value, overwrite currentC with the writeCertificate and reply Write2Ans with updated object value. These rules provide determinism when multiple commits happen within one epoch.

4.2.2 Epoch exhaustion

It is possible for too many concurrent writes exhaust 1000 range of numbers dedicated to it. If that happens, the clients can identify that and execute special protocol procedure which will start NewEpoch, without modifying current one.

4.3 Garbage Collection

MochiDB produces a lot of stored metadata (notably given Write1 multigrants) for every object in the datastore. To save disk space and prune expired Write1Grants, MochiDB uses Garbage Collection (GC). When performing GC, MochiDB treats objects independently. Each server runs a background thread that removes Write1 multigrants that were given 2 or more epochs before the currentC's epoch. Recollect that we save Write1 Grant in server for every Object to avoid giving new

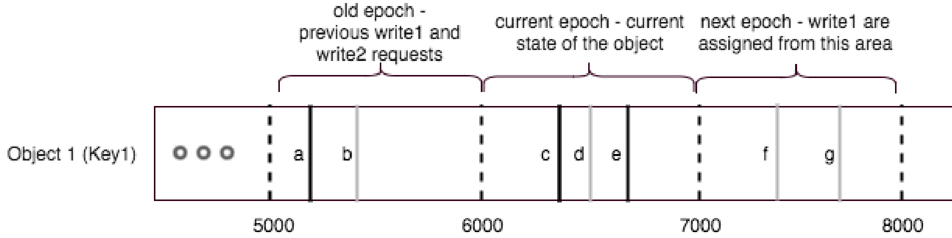


Figure 3: Object Epoch Timeline

Object 1 has current epoch 6 (6*xxxx*) because during that epoch the last Write2 happened - points *c* and *e*. During epoch 6 Write2 (point *d*) was granted but never finished. Epoch 5 (5*xxxx*) is old epoch with one finished Write2 (point *a*) and one non finished (point *b*). New epoch is 7 (7*xxxx*). Timestamps for write1 requests will be granted from that epoch. Currently two grants happened (points *f* and *g*), but no Write2 was received yet.

Write1 Grant to a different transaction on the very same timestamp and also to respond to Write1ToServer retries. As we have moved to advanced epochs, the old Write1 multigrants are no longer needed as they had completed their purpose of existence.

4.4 Resync

Clients can detect slow mochiDB servers when they consistently issue Write1 multigrants that are from older epochs from the rest of the cluster. When alterted by a client with an *UptoSpeed* operation, the slow mochiDB server acquires write locks on those objects so that all read, write operations pertaining to them stall and proceeds to execute a special sync operation with an upto date mochiDB server identified by the client. This sync operation involves overwriting the identified set of objects' value, currentC and given Write1 MultiGrants to match those from the upto date server to complete the resync.

4.5 Sharding

MochiDB supports sharding natively. Each key is mapped to some hash by known and fixed hash function. All of available space is divided into equally sized partitions using Q tokens. Since hash space is known and the number of partitions fixed, its trivial and deterministic to calculate those tokens. Each server is assigned multiple tokens by administrator depending on server performance, location to readers/writers, etc. That mapping is stored in the configuration of each server and also can be reliably retrieved by the client. When data is stored, it is stored on N nodes along the ring starting from the next token on the ring. The algorithm to determine nodes is the following:

1. Apply hash to the key and get H_k
2. Select the next token on the ring which follows or equals to H_k
3. Circle the ring clockwise and select N nodes. Those will be the servers participated in transaction.

Due to BFT requirements of the protocol, extra restrictions are added to the process above. N selected nodes = 3 f + 1, where f is number of faulty replicas. That has the following implications: If we want to double capacity (i.e. reduce by half number of partitions/shards on each server), in order to maintain the same f, we must double number of servers. Or if we reduced by half number of partitions on each server with the same amount of servers, we are also effectively reducing f.

Servers which are mapped to N selected tokens must be unique. That prevents the same faulty server participating multiple times in the transactions. That is fixable at the assignment time - when tokens are assigned to servers no 2 out of N sequential tokens are given to the same server.

4.6 Configuration changes

MochiDB allows configuration changes without reboot. Configuration is stored similar to other keys and all configuration keys starts with "CONFIG_". We introduced configurationstamp (CS) which is a number, that increments every time configuration changes. During all operations (such as write1, Write2, read, gc, etc.) CS is being passed alongside the message. The server denies message processing if its current CS differs from the received CS.

The following algorithm is executed on a client with admin privilege:

1. Administrator allows all outstanding UpToSpeed actions to completion
2. Administrator sends config1 message to ALL servers. When servers receive config1, after validation they perform the following:
   (a) Checks whether there are concurrent config1 messages being processed at the moment. If there is one, the server will respond back with

error.
   (b) Blocks server from accepting any new read, write messages.
   (c) Send back config1ready
3. The client waits for majority of config1ready messages, creates configChangeCertificate and proceed to phase 2.
4. During phase2, client send configChangeCertificate to each of the servers through config2 message. Upon receiving of message, each server will apply configuration, increment CS and reply with ack.
5. Configuration change assumed to be complete when client receives majority of acks.

5 Implementation

We built MochiDB using Java8, Netty as asynchronous network communication library, Protobufs as serialization library, Spring Boot for REST API and management UI. The functional, test and deployment code is over 48,000 lines and is publicly available in our open repository[5]. We used in memory DB, but switching to some other DB (like MySQL) in future should be simple and uncomplicated. We have also published our in memory DB, mochiDB docker image [4] for the public to test and deploy them in public cloud and enterprise data centers. Our implementation lacks PKI support - that work is left for the future, but our tests showed that adding TLS and signing messages should not have huge impact on performance.

5.1 Evaluation

During development we spun up virtual mochiDB server and client clusters within the JVM and built a test framework that tested the underlying sharding scheme, read, write, delete operations, concurrent transactions and a special stress test scenario. To evaluate the performance on a WAN setup, we devised 5 mochiDB clients concurrently executing transactions with a 5 node mochiDB Server cluster (sharded) running on AWS public cloud, us-west-1 zone. The average ping time between client and server clusters is about 13 ms. Each local client gets assigned 40 distinct keys and performs the following sequence of transactions : write key-value, then read each key and verify the content, followed by deletion of all keys. Table 2 on page 6 describes read, write times measured. During this stress test, distinct keys were assigned to clients. We noticed higher latencies for write and reads, when test clients overlap keys.

Table 2: MochiDB WAN performance

READ	Time	WRITE	Time
50th percentile	26.6 ms	50th percentile	56 ms
95th percentile	31.1 ms	95th percentile	98 ms
99.9th percentile	33.9 ms	99.9th percentile	145 ms

6 Optimization

We have identified that using leases will greatly improve write performance across the cluster and the use of hashes to reduce size of messages will improve the overall throughput. We can minimize metadata overhead, by only storing Grants pertaining to an object from majority of MochiDB servers instead of the all encompasing *writeC*, that contains majority grants for all objects involved in the transaction.

7 Conclusion

In this paper we presented MochiDB - distributed, consistent, BFT key value store which is intended to work over high latency communication network. Our testing showed that MochiDB is a viable product that can operate well under high read requests and moderate write requests. Our work is the first attempt to build a production ready, WAN distributed BFT datastore for configuration management. In building MochiDB, we applied several enhancements to quorum based protocol including introduction of random write seeds and epochs to minimize contention resolution costs on the server.

References

[1] CASTRO, M., AND LISKOV, B. Practical byzantine fault tolerance. *Proceedings of the Third Symposium on Operating Systems Design and Implementation* (1999).

[2] JAMES COWLING, DANIEL MYERS, B. L. E. A. Hq replication: A hybrid quorum protocol for byzantine fault tolerance. *USENIX Symposium on Operating System Design and Implementation* (2006).

[3] Ping Latencies across the globe. `https://wondernetwork.com/pings`. Accessed: 2017-11-26.

[4] TSATURYAN, T., AND DHAKSHINAMURTHY, S. DockerHub. `https://hub.docker.com/r/mochidb/mochi-db/`.

[5] TSATURYAN, T., AND DHAKSHINAMURTHY, S. Repository. `https://github.com/saravan2/mochi-db/`.