MaoBFT: A Single Leader BFT Transaction System

Weilun Chen Boning Gao

***Abstract*— Among Byzantine fault-tolerant protocols, one of the most expensive operations is master re-election. While in the real world, leader re-election could be unnecessary. For example, if the leader is a well-known company such as Google or Visa, it would hardly fail and even it fails it usually comes back online very soon. Another scenario is a system with strict financial regulations, a fixed leader is needed to detect malicious transactions.**

In this paper, we present MaoBFT, a fixed leader BFT Transaction system that runs on top of async network. We sacrifice liveness when the leader fails to achieve higher throughput. Also, using an optimized reliable broadcast algorithm, we reduce the communication complexity to $O(N)$. Finally, we present a blockchain-based synchronization mechanism that greatly reduces tail latency in async network. It allows slow nodes to catch up with peers by actively pulling missing data rather than purely relying on async network's "eventual delivery" guarantee.

I. INTRODUCTION

Distributed System is the de facto solution for an ultra-scale software system that grows beyond a single machine's capability. Almost all of our daily interacted systems are distributed in some way. Distributed System has the imperial benefit of tolerating single point failure. When the master is down, usually a leader re-election process will happen and a new leader will be online and coordinate the cluster. However, on the other side of the coin, leader re-election is usually computation and communication intensive. For example, RAFT requires a timeout to kickoff leader reelection and is not guaranteed to finish in a single round. It creates $O(N^2)$ messages in the network, and processing these messages requires computation resources. This problem would further be compounded with Byzantine general problem. In PBFT, a view change requires complex exchanging of local uncommitted pre-prepare messages.

In some real-life scenarios we can sacrifice liveness when a leader encounters a recoverable failure. For instance, in the case that leader is a well-known party such as VISA, we can assume that leaders hardly fail and will be back online soon when a blackout happens. Moreover, in the case where the leader is an authority such as the government, the authority should hold the lead all the time. With this observation, our first objective is **light weight**. Our goal is to develop a broadcast algorithm that doesn't require leadership change and runs in the scenario where occasionally leader failure is acceptable. This greatly simplifies the design and reduces the communication burden of leadership handling. It's worth pointing out that the leader node can be an abstract node that consists of multiple physical machines. There could be another consensus protocol running among these nodes such as RAFT, the specific algorithm is out of the scope of this paper.

With the unprecedented success of Bitcoin, the Byzantine general problem is back on the table again. In this setup, instead of fail-stop, a misbehaving node can act arbitrarily during failure. Furthermore, we also make a loose assumption about our network. We assume the network to be *asynchronous*, where the only guarantee is eventual delivery. With those 2 challenges, our second objective is to implement our broadcast system the runs on **async network** and can tolerate **Byzantine failure**.

Various works have been proposed to solve the problem of BFT broadcast on async network. Some notably include Reliably broadcast, where at most f nodes can be Byzantine nodes in a $N = 3f+1$ cluster. However, this algorithm relies heavily on async network's guarantee. The problem of solely async network guarantee is that there is no secured delivery time. In the case where messages need to be serialized, this could create a problem that fast messages be blocked by slow ones in earlier positions, creating a long tail in transaction delivery. Thus, our last goal is to resolve this long-tail latency problem by introducing a way to $synchronize$ for slower nodes to catch up. This cannot be achieved by asking any peer directly because there is no way to know whether the answer is correct or made up by a Byzantine node. However, with the emerging of blockchain, we can rely on it to safely synchronize with peers if we partially know the chain and just need to fill holes in the chain. We'll describe in detail in later sections.

TABLE I
OBJECTIVE OF MAOBFTS

Light Weight	MaoBFT sacrifices liveness when master is down.
BFT on Async Network	MaoBFT runs on async network and can tolerate at most f Byzantine nodes in a $N = 3f + 1$ cluster
Fast Sync	Slow nodes can catch up by syncing with any peer

Finally, to demonstrate our design, we implemented a transaction system. Users can deposit money or transfer from one to another. We choose a transaction system because it requires strong ordering that a later transaction cannot be applied if the previous ones haven't been applied yet. Strong ordering will help us demonstrate the benefits of having a synchronize mechanism since later transactions will all be blocked if previous ones are not received yet.

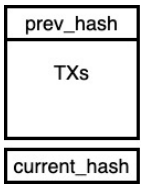

Fig. 1. Block

Our paper expands as follows. We present related prior works in section ii. Section iiiexplained our system implementation. Section ivpresents our evaluation and in section vwe'll explain future works to augment the system.

II. PRIOR WORK

In the case of a Byzantine fault, a server can appear failing and functioning at the same time, behave differently to different observers. It is difficult for the other components to define its status and kick it out of the cluster because they need to first reach a consensus on which component has failed in the first place.

PBFT[1] is the fist well-studied protocol to solve this category of problem that tolerates optimal of f Byzantine nodes in an $N = 3f + 1$ nodes scenario. PBFT requires replicas to first agree on a unique, serial ordering of requests; the requests are then executed in that order and replies sent to the clients. However, PBFT is based on a weak synchronous assumption that message delay $d(t)$ doesn't grow faster than time t indefinitely. Weak synchronous assumption can be easily reverted it the opponent gains control over the network schedule. To address this, HoneyBadgerBFT[2] makes async network tolerable by using an asynchronous common subset(ASC) with reliable broadcast(RBC). HotStuff reaches linear view change and also reduced the authenticator complexity. In this paper, we want to focus on a single leader scenario so view change cost is not a concern.

RBC was first introduced by Bracha[4]. It is an asynchronous byzantine agreement protocol with communication complexity of $O(n^2)$. C. Cachin and S. Tessaro reduced the complexity to $O(n)$ with erasure codes[6]. Although RBC has a higher authenticator complexity[3], we valued its asynchronous property which can benefit the throughput and system robustness. So instead of using signature combining via threshold cryptography in HotStuff way, we adopted an optimized RBC as a building block.

III. IMPLEMENTATION

In this section, we'll talk about how do we implement our system. In the section below we refer to the broadcaster as *leader* and nodes that only participate as *follower*, we also refer to users that try to commit information into the system as *client*. For its fast runtime speed as well as understandability, we choose Golang to implement our system. Messages communicated between nodes are defined using protobuf, and we implement the communications channels between nodes using gRPC.

Overall, our system consists of 3 layers that perform different functionalities:

RPC layer: In this layer, a node exposes a bunch of different RPCs to clients as well as peer nodes. Note that all clients make requests to the leader, and leader exposes 2 interfaces to clients:

- **ProposeTransaction(ProposeTransactionRequest)**: This RPC service allows users to propose a transaction that changes the state of the system. There are 2 types of transactions supported: Deposit or Transfer. A deposit transaction can only be issued with a special client called administrative client, while transfer transaction can be proposed by any client. We validated their identity and information integrity by signing a digest of the information. A UUID will be returned to the user, and the user can obtain the transaction status by looking it up through **GetTransactionStatus** RPC.
- **GetTransactionStatus(UUID)**: By providing an UUID obtained from **ProposeTransaction**, a user can get the status of transaction. There are 5 possible transaction statuses: ***COMMITTED***: the transaction is accepted and applied to the system. ***STAGED***: the transaction is accepted but not applied to the ledger yet. ***PENDING*** means the transaction is created by the leader and broadcasted out. ***REJECTED*** means the transaction is either not a valid transaction, or somehow didn't make it to the system due to network partition. Finally, ***UNKNOWN*** means this transaction is only received by the leader but not processed yet.

RPC layer also maintains an event queue. We batch these user transactions to reduce the system overhead. An event queue caches client requests in FIFO order based on its arrival time. It rejects any order that is invalid to the system such as overdraw. We guarantee that the order in the queue will also be the same order that will get committed to the ledger.

Note that, periodically, the leader will send out an empty request to followers if no request has been sent out beyond a time t. This heartbeat is to help followers sync up with the system, which we'll describe in detail later.

Application Layer: We maintain the ledger and validate transactions in memory in this layer. The ledger we maintain is essentially a hashmap that stores account names to their balances. When receiving a message coming from the broadcast channel, we validate it with the ledger and discard invalid messages. For example, if a transaction in the message overdraws one's account, we will not apply the entire message. Note that this layer is not persistent, and we defer persistent responsibility to the blockchain layer that we'll describe very soon.

Core Layer: The core layer consists of 2 components: Blockchain and RBC.

We use blockchain to store all messages, a.k.a Blocks, in a persistent way to handle failover. It also enables synchronize mechanisms that couldn't be achieved using a normal logger. A block will contain multiple transactions to reduce system overhead, and linearizability is achieved that transactions in

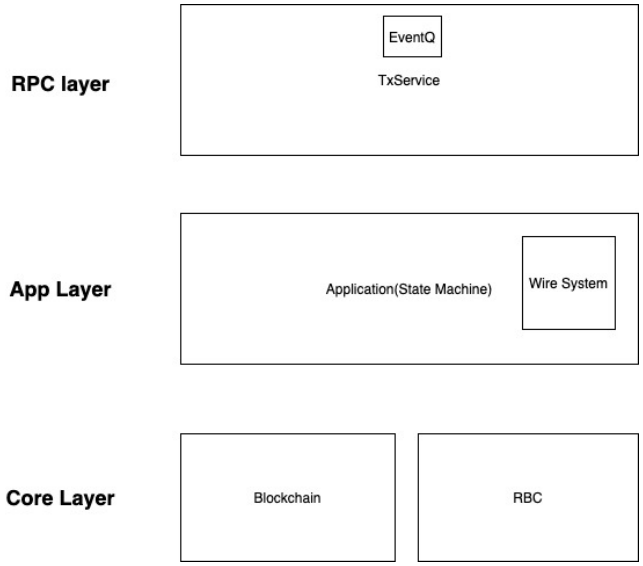

Fig. 2. Overall Architecture

blockchain will be the same order as added to the event queue. Another core component is RBC module, which takes one or multiple blocks and reliably broadcast it in the underlying async network. We implement the async network by infinitely retrying with exponential backoff until we succeed. We'll show in later sections how Synchronize Mechanism can increase RBC throughput.

A. Optimized Reliable Broadcast

We made some modification to RBC, the algorithm is shown in Fig.3.

A big difference is that we make blockchain a building block of RBC. It helps us add SYNC message to reduce the time for a fallen behind node to catch up and increase the system throughput(see Fig.6). SYNC mechanism details will be explained in the following subsection.

But blockchain brings in a split-brain case, when a byzantine leader sends two blocks v_1, v_2 with the same prevhash in a manipulated order that some followers committed v_1 first and others committed v_2 first. When the other block came, the slot on the blockchain is already taken which results in a failure commit and two different chains. To avoid this split-brain case, PREPARE carries prevhash pv of block v and followers will verify a different block has not been received with the same pv before multicasting it.

B. Blockchain Design and Wire System

As demonstrated in 1, each block consists of 3 sections. A $prev_hash$ which is the SHA256 hash of the previous block's content. Transactions TXs proposed by clients, and cur_hash which satisfies:

$$SHA256(prev\_hash, TXs) = cur\_hash$$

We partition the blockchain into 3 areas demonstrated by 4. Specifically, there are 3 areas in the blockchain:

Optimized RBC(leader$P$, with followerP_i)

- upon client request(v) where v is a block with hash h and prevhash pv,
  let $\{s_j\}_{j\in[N]}$ be the splits of an $(N-2f, N)$ erasure coding scheme applied to v
  let h be a Merkle tree root computed over $\{s_j\}$
  P send PREPARE(h, b_j, s_j, pv) to each party P_j, where b_j is the j^{th} merkle tree branch
- upon receiving PREPARE(h, b_j, s_j, pv) from P,
  if pv has been received with a different Merkle root of b_j, discard
  multicast ECHO(h, b_j, s_j, pv)
- upon receiving ECHO(h, b_j, s_j, pv) from P_j,
  check that b_j is a valid Merkle branch for root h and leaf s_j, and otherwise discard
- upon receiving valid ECHO$(h, _, _)$ messages from $N - f$ distinct parties,
  - interpolate $\{s^{j}\}$ from any $N-2f$ leaves received
  - recompute Merkle root $h^{'}$ and if $h^{'} \neq h$ then abort
  - if READY(h) has not been sent, multicast READY(h)
- upon receiving $f + 1$ matching READY(h) messages, if READY has not yet been sent, multicast READY(h)
- upon receiving $2f + 1$ matching READY(h) messages,
  wait for $N - 2f$ ECHO messages, then decode v
- Apply block v to application
  - if block v's previous block is *COMMITTED*, COMMIT(v)
  - otherwise STAGE block v
  - if 3 or more blocks are *STAGED*, multicast SYNC(h_l, h_f) where h_l is the hash of last committed block and h_f is the hash of first staged block

Fig. 3. optimized reliable broadcast

Committed: The blocks in this area are connected, meaning every block's $prev\_hash$ is the previous block's cur_hash. A Block gets committed into blockchain if it is valid and received by RBC described above.

Staged: A block will be added to the staged area if it's received from RBC, but not able to connect to previous blocks in the committed area.

Pending: A pending block is added only in the leader's blockchain. And it contains transactions that are broadcasted by this leader, but no consensus is reached yet.

Our blockchain is persistent by using on-disk persistent storage, and we assert blockchain as the only source of truth for recovery after failover.

As for the wire system, we maintain it in memory. It will validate blocks in the Committed chain by iterating every transaction within that block in order. If any transaction is invalid, we abandon the entire block and don't apply it to the wire system. Note that the internal state of the wire system can be recovered by simply reading the block in order during

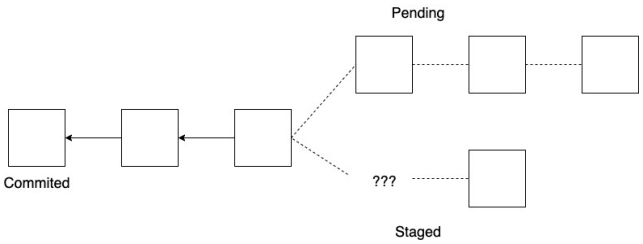

Fig. 4. Blockchain Design

failover, and we don't need to dump the data inside the wire system into a disk.

C. Syncrhonize

Solely relying on async network's promise can result in a very long backlog. For instance, all blocks in staged area could be prevented from committing by just one single block, and there is limited things we could do rather than just wait. We tackle this problem by letting nodes to actively asking for missing blocks if the backlog it too long. To achieve this, one way of doing this is to ask at least $2f+1$ peers about the missing blocks. However, we observe that with blockchain, we could simply ask any peer to help fill the gap. The nodes that needs to sync will create a **Sync** request composed of both $lastCommittedBlock$ and $latestStagedBlock$. And any peer that provides a valid answer can be trusted and adopted immediately. This is based on 3 facts:

- Any time we receive a block from RBC, there could be an opportunity window that we can retry.
- $lastStagedBlock$ and $lastCommittedBlock$ received by any non-faulty node will be eventually received by all non-faulty nodes.
- No one can fake a blockchain answer since it's considered as computational impossible.

Thus, in our system, any follower that have long staged chain can issue a best effort **Sync** request to its peers in round robin order. A non-faulty node will respond to this request, if it can answer, by providing a list of blocks that fill the gap between $lastCommittedBlock$ and $latestStagedBlock$. In practice, we set the threshold to sync as more than 3 blocks in the staged area. To prevent stagnation if no request is issue in a period, leader will issue an empty block to followers as a heartbeat. This block can trigger the sync mechanism and keep every node in the same page.

IV. EVALUATION

We evaluated our Mao-BFT implementation[7]. All tests ran on a MacBook with 2.9 GHz Intel i7 CPU and 16 GB memory. Since we didn't deploy it to cloud servers, the network can be neglected. This evaluation reflects the computational costs of messages.

A. Latency

Transaction latency is the time duration after a request is sent until committed when the leader receives $2f+1$ READY and $N-2f$ ECHO. The latency number is an amortized result generated with the Go benchmark framework.

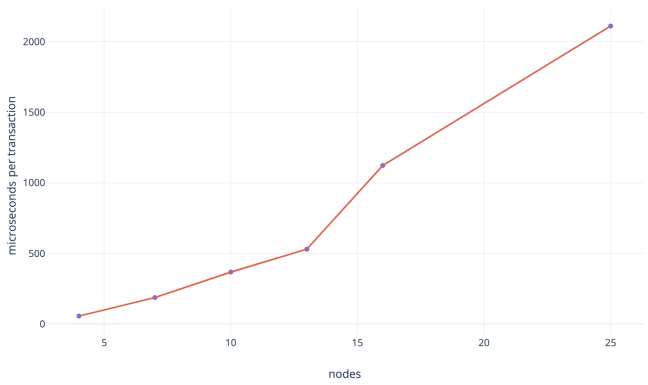

Fig. 5. transaction latency and number of nodes

As shown in Fig.5, the amortized latency grows linearly with the number of participants. Erasure coding and Merkle tree reduce the communication cost of RBC to O(n).

B. Throughput

The pure async network is hard to mock. In our system, each message is an async call with infinite retries in case of failure. However, gRPC's connection establishments are expensive. So we came up with a Sync mechanism with exponential backoff timeout to avoid frequent connection creation. To trigger retries, we deliberately pull one server down. So this benchmark runs with 1 leader and 2 followers.

Fig. 6. throughput comparison

The blue line in Fig.6 shows the throughput with the timeout set to a fixed 1 second. The red line shows the throughput when exponential backoff timeout is applied(retry after 1s, 2s, 4s until 300s as maximum). When the timeout is set to 1 second, the connection reestablishment overhead can slow the system badly and the throughput is 53 transaction/second. Also in this fixed timeout setting, the more messages sent, the more retrying threads will be running, so the curve becomes steeper as more messages sent out. With exponential backoff(red line), the throughput reaches

263 transactions/second with a flatter curve indicating the throughput is more stable.

V. FUTURE WORK

To increase the leader's availability, the leader can be a Raft-based cluster, which is not implemented yet. Leader reelection can happen within this Raft cluster, which is much cheaper than happening in the whole BFT system. And there are other combinations of consensus protocols besides Raft and RBC. An alternative to RBC is signature combining via threshold cryptography, where a leader requires a larger bandwidth to send requests and collect responses from all followers.

Dynamic membership is another feather we put the effort into but decided not to include. It is not easy to add a node dynamically to an RBC protocol. Because each participant needs to update its byzantine limit and authorized keys, which can be blocked by potential byzantine participants. If we make the condition looser, ex. a certain quorum of acknowledgments are needed, the system is possible to end up in a split-brain state. A straightforward solution is to shut down all nodes and reconfigure then bring them back online together.

VI. CONCLUSION

Mao-BFT is a single leader Byzantine fault-tolerant system. It sacrifices liveness when the leader fails but achieves a simpler design and higher throughput if the leader hardly fails. It fits in scenarios where financial regulations are strict and a fixed leader can detect malicious transactions. It can also be utilized by a company that provides servers with high availability to reduce the overall cost of a BFT transaction system.

REFERENCES

[1] M. Castro, B. Liskov, et al. Practical byzantine fault tolerance. In OSDI, volume 99, pages 173–186, 1999.

[2] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. 2016. The Honey Badger of BFT Protocols. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016. 31–42. https://doi.org/10.1145/2976749.2978399

[3] Yin M, Malkhi D, Reiter M K, et al. Hotstuff: Bft consensus with linearity and responsiveness[C]//Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing. 2019: 347-356.

[4] Bracha G. Asynchronous Byzantine agreement protocols[J]. Information and Computation, 1987, 75(2): 130-143.

[5] Nakamoto S. Bitcoin: A peer-to-peer electronic cash system[R]. Manubot, 2019.

[6] C. Cachin and S. Tessaro, "Asynchronous verifiable information dispersal," 24th IEEE Symposium on Reliable Distributed Systems (SRDS'05), Orlando, Florida, USA, 2005, pp. 191-201, doi: 10.1109/RELDIS.2005.9.

[7] Mao-BFT https://github.com/gopricy/mao-bft