BlockVote - A Blockchain-Based Digital Voting System

Zhongze Chen, Eric Lyu, Mingyang Ye, Xinyi Ye, Yitong Zhao
Github Repo

1 Introduction

Democratic elections have relied on paper-based voting systems. While the system works well enough for most elections, it still suffers from the inherent weakness of the centralized trust-based model, as tallying paper votes requires a trusted administrative system. Driven by the desire to make the voting system more secure, verifiable, transparent and efficient, electronic voting or e-voting evolved. In the past decades, e-voting systems adapting to internet technologies have achieved remarkable progress [2]. This paper proposes implementing a decentralized e-voting system utilizing blockchain technology.

Blockchain technology has been adopted for many years to remove trusted arbitrators from a system and prevent disputes. Nakamoto [3] established Bitcoin, an electronic cash system utilizing blockchain to verify transactions and prevent double-spending. We base our blockchain structure and PoW on Nakamoto's article.

Since blockchain is a P2P system, it requires a way for peers to communicate information such as transactions and blocks so that every replica receives every update eventually. Gossip Protocol is a P2P communication algorithm based on how an epidemic spreads [1]. In Gossip Protocol, every node in the system will randomly select another node and resolve any differences between the two nodes by exchanging information. Eventually, all nodes will receive the latest piece of information. This protocol has been widely adopted in P2P systems as it is fast, scalable, highly available, and fault-tolerant. In our BlockVote implementation, we utilize Gossip Protocol to broadcast transactions and blocks to peers.

2 Design

2.1 Overview

Our system will implement an e-voting system as a blockchain that is distributed among all nodes. The system consists of a coordinator (coord), at least one miner and some clients, as shown in Fig. 1.

- **Coord:** initializes the system and coordinates nodes in the system
- **Miner:** validates transactions by mining new blocks to be added to the blockchain
- **Client:** provides e-voting services for voters by sending votes as transactions and querying voting results

Clients are designed to be the voting stations that are capable of providing services for multiple voters. Voters cast their votes with their signatures at their designated clients in the form of transactions. Transactions are submitted to

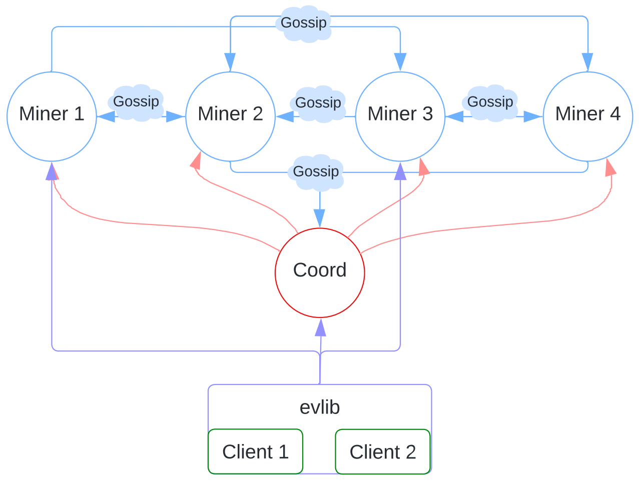

Figure 1. System architecture

miners to be validated. Miners will show their acceptance of the transactions by including them in the next block they are going to mine. They will race to include transactions in the block and chain the block into the blockchain using proof-of-work. Coord and miners will communicate transaction and block information via Gossip protocol. Our system will be available when the coord and at least one miner are alive.

2.2 Assumptions

- **System**
  - All nodes in the system will not have any malicious behaviour.
  - Coord and miner failures can be detected by fcheck or the failure of a single RPC call.
  - Network may be unreliable but there is no network partition.
- **Coord**
  - Coord will be started before miners and clients.
  - Coord may fail but will recover.
  - Coord will not fail during miner joining.
- **Miner**
  - Each miner can be identified by a unique miner ID and address.
  - After the first miner joins the system, at least one miner is alive.
  - Miner may fail at any time after joining and will not recover.
  - At any time, miner will not have more than 40 * *MAX_TXN* pending transactions.

- **Client**
  - Each client can be identified by a unique client ID and address.
  - A client have multiple voters, and each voter can be identified by a unique pair of the private key and the public key. Voters of different clients will not overlap.
  - Client will not start until the first miner joins the system.
  - Client can only issue a new transaction after the previous transaction has been submitted to selected miner.
  - Client will never fail.

2.3 Voting Rules

We consider the following voting rules as part of our system:

- Votes with signatures that cannot be verified are invalid.
- A voter can only cast one vote. If a voter cast more than one votes, only one vote is valid.
- A voter can only vote for candidates. If a voter votes for non-candidates, the vote is considered invalid.
- Candidates are not allowed to vote. Votes from candidates are invalid.

2.4 Structures

Our system will have the following structure:

- Coord initializes the blockchain by generating the genesis block.
- Miners join the system by first contacting coord for a copy of the blockchain, candidate list and peer list. Then they connect to a random peer to download the pending transaction. Finally, they register themselves as miners at coord to complete join.
- Clients contact coord to retrieve the candidate list and miner list at startup.
- Client creates a transaction with ballot data and submits it to a randomly selected miner.
- Miner put transactions in the memory pool and broadcast them by gossiping upon receiving.
- Miners select up to *MAX_TXN* transactions based on the time of arrival from their memory pool and validate them. Then they will include validated transactions in the next block to be mined.
- Miners will compete to be the owner of the next block in the blockchain by solving the proof-of-work puzzles.
- The miner who mines the new block will gossip about the block to randomly selected neighbours. Neighbours will also gossip about the new block to their neighbours until the new block completely propagates the entire network.
- Miners receive and validate the new block and add it to their local copy of the blockchain. If the new block is part of the longest chain, they will stop their work and build on the new block.
- The system considers a transaction to be committed if the block that contains the transaction is (1) part of the longest chain and (2) confirmed by *NUM_CONFIRMED* blocks mined since.

2.5 Properties

Our system provides the following guarantees:

- **Valid transactions** will eventually be committed
- **Invalid transactions** will not be committed.
- Any group of **conflict transactions** (transactions that are from the same voter but for different candidates) will have exactly one committed. In other words, the system will not have any conflict transactions.
- Our system will be available when the coord and at least one miner are alive.

3 Implementation

We implement our system as five parts:

- **Identity** implements digital wallet to provide a pair of public key and private key as identity for each voter.
- **Blockchain** defines the data structures for ballot, transaction and block and provides all blockchain functionalities.
- **Gossip** implements the gossip protocol for transaction and block broadcasting.
- **BlockVote** implements core logic for miners and the coord.
- **E-voting Library (EVlib)** implements APIs for clients.

3.1 Identity

The identity package provides the supports for uniquely distinguishing each individual from others. The identity can be divided into 2 types, either as a voter or a candidate and can be created through the following function.

- `CreateVoter(name string, id string) (*Wallets, error)` – creates a voter in respect to the corresponding info, returns voter's wallets
- `CreateCandidate(name string) (*Wallets, error)` – creates a candidate in respect to the corresponding info, returns candidate's wallets

Note that the `CreateCandidate` function will only be used at the initialization before the actual voting starts. For each Wallets returned, it contains the following fields.

- `UserType string` – specifies the type of the user, can be either as the voter or the candidate
- `Wallets map[string]*Wallet` – mapping of the encoding address to the wallet
- `VoterData Voter` – stores the information about the voter's name and the student ID. It will leave as an empty struct if user is a candidate.

- `CandidateData Candidate` – stores the information about the candidate's name. It will leave as an empty struct if user is a voter.

Each individual Wallet will consist of a pair of public key and private key. The key pair will be generated through the P256 elliptical curve of the encrypted package `ecdsa`. To make the system more robust, we save the information of identity under the **./tmp** to avoid losing ground truth of individual's existence.
Essentially, each identity may only need one wallet. However, one may have multiple wallets which are used in multiple different voting. In order to align with the standard introduced in Bitcoin, we incorporate hashing of the public key into the address which used in the mapping of address to wallet. We temporally make the size of Wallets map to be 1 which means one can only have a wallet for now. In the future, it may be scaled for multiple wallets.

3.2 Blockchain

3.2.1 Ballot. The basic structure we used for storing the information of a vote is Ballot. Essentially, a ballot would imply a vote is going from the voter with its name and student ID to one candidate and also means the number of votes from such the voter to that candidate is **one**. Below is the struct of the ballot.

- `VoterName string` – the voter's name, should be globally unique and appear just once since a voter can only have one vote.
- `VoterStudentID string` – the voter's student ID, should appear with the voter's name.
- `VoterCandidate string` – the candidate's name which specifies where the vote goes to.

The ballot under the context of our application will be served as a basic unit of data that goes with the transaction as it has been created, signed, etc. Also, the data of the ballot also be utilized for the hashing of fields like **TxID**.

3.2.2 Transaction. A transaction is created in `EVlib` whenever a client wants to vote.

The `Transaction` data structure in our system includes the following fields:

- `Data Ballot` – the voter's ballot
- `ID []byte` – transaction ID, the hash of `Data` and `PublicKey` field, uniquely identifies a transaction and can be used to query the blockchain
- `Signature []byte` – signed transaction ID using the voter wallet's private key, and will leave `nil` if the transaction not signed
- `PublicKey []byte` – public key of the corresponding voter's wallet, for miners to validate the signature

For each `Transaction`, we provide functions for voters to sign the transaction and for the blockchain to verify the transaction as the following:

- `Sign(privKey ecdsa.PrivateKey)` – signs the transaction ID with the voter's private key using `ecdsa.Sign`, and stores the returned signature in the `Signature` field. Both the `Signature` and the `PublicKey` will be used to verify this transaction by miners.
- `Verify() bool` – verifies the transaction's `Signature` using `ecdsa.Verify`. Underneath it derives the transaction hash from the `Signature` and `PublicKey`, and compare this hash with the hash directly computed from transaction data. If the two hashes match, the transaction is verified.

3.2.3 Block. The `Block` data structure in our system includes the following fields:

- `PrevHash []byte` – the hash of previous block in the blockchain
- `BlockNum uint8` – the height of the block
- `Nonce uint32` – 32-bit unsigned integer that is calculated by the PoW algorithm, with initial value 0.
- `Txns []*Transactions` – transactions mined in the block
- `MinerID string` – ID of the miner that mines the block
- `Hash []byte` – the hash of all the contents in the block

When a block is created by a miner, its `Nonce` is initialized to `0` and the `BlockNum` will be set to the `BlockNum` of the previous block plus 1. The block's `Nonce` is calculated by calling `NewProof` and `Run` functions of `ProofOfWork` (see Section 3.2.4).

Our `Block` structure implements the genesis block, which is the very first block of the blockchain. The Genesis block's previous hash field is empty and and its height is set to be `0`. It's owned by the coord and does not contain any transactions.

3.2.4 Proof of Work. The PoW algorithm iteratively calculates the nonce of a block that makes the block hash have the required number of leading zeros.

Our `ProofOfWork` structure has the following field:

- `Block *Block` – the block whose nonce is to be found
- `Target *big.Int` – a target number we compare the block hash with to check if the correct nonce is found

We implement the following functions for the `ProofOfWork` structure:

- `NewProof(b *Block) *ProofOfWork` – creates a new `ProofOfWork` structure for a block, sets the `target` to have `NumZeros` leading zeros
- `Run()` – execution of proof of work, keeps calling `Next` until it finds a nonce that makes the block hash has `NumZeros` leading zeros
- `Next(delayed bool) bool` – returns true if the block hash is less than the target, otherwise increments the block `Nonce` by 1 and returns false

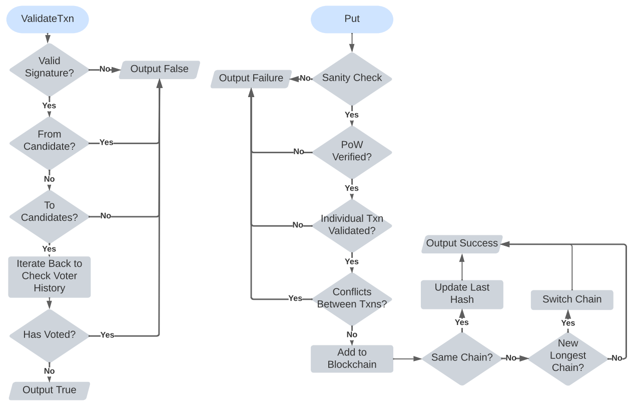

Figure 2. Workflow of ValidateTxn() and Put() in Blockchain

- `Validate() bool` – checks whether a block `Nonce` is correct by hashing the block and comparing the hash with the `target`
- `BlockToBytes(nonce uint32) []byte` – converts the block contents to bytes so that the block can be hashed by `sha256.Sum256`
- `NumToBytes(num uint32) []byte` – converts a `uint32` to bytes, used for block `Nouce` and `BlockNum`
- `HashTxns() []byte` – uses `sha256.Sum256` to hash block transactions
- `EncodeTxn(tx *Transaction) []byte` – converts each block transaction to bytes so that it can be hashed by `HashTxns`

3.2.5 Blockchain.

Blockchain has information of all the blocks mined, including uncle blocks and competing chains. To efficiently store and retrieve blocks, we use a key-value database called BadgerDB as the underlying storage for our blockchain. To store a block in the DB, the block hash prepended by a constant prefix string is used as the key and the encoded block is the value. We also implement an iterator `ChainIterator` to iterate from the tail of the blockchain to the head.

The `Blockchain` structure has the following fields:

- `LastHash []byte` – hash of the last block on the longest chain
- `DB *util.DataBase` – DB instance for read and write
- `Candidates []*Identity.Wallets` – all the candidates in the voting system

The operations we implement for the `Blockchain` structure are:

- **Blockchain Initialization** – `Init()` initializes the blockchain with the genesis block. It will first generate a genesis block and store it in the DB. Then the `LastHash` of the blockchain is updated to the hash of the genesis block.
- **Transaction Validation** – `ValidateTxn()` checks if a transaction is valid. The work flow of transaction validation is shown in Fig. 2. First it calls `txn.Verify()` to verify the signature of the voter. Then it checks the transaction based on the criteria described in Section 2.3. Public key is used to check if a voter is a candidate. If the voter's public key is the same as one of the candidates', then the transaction is invalid. The `CandidateName` field in the ballot will be used to check if the voter votes for a non-candidate. If the candidate's name does not match any known candidates, the transaction is invalid. To check if a voter has voted before, it iterates through the longest chain to find if there is existing transaction signed by the voter. If such transaction is found, then the transaction is invalid. We also implements `ValidateTxns()`, which checks if a set of transactions are valid. It validates the transactions one by one, and it also makes sure no voter in this set of transactions votes more than once.
- **Transaction Inspection** – `TxnStatus()` returns the number of blocks that confirm the given transaction. It iterates through the longest chain and counts how many blocks follow the one containing the given transaction. If the transaction is not found, it will return -1.
- **Results Inspection** – `VotingStatus()` returns the number of votes each candidate gets and corresponding transactions. It iterates through the longest chain, but skips the last *NUM_CONFIRMED* blocks as a transaction is only committed if there are *NUM_CONFIRMED* following blocks. Then for each block, it iterates through the transactions in the block, add each transaction to the `txns` array, and increment the number of votes of the corresponding candidate based on the candidate name in the transaction data.
- **Block Retrieval** – `Get()` gets a block from blockchain by hash. It constructs key from the hash, and retrieve the corresponding block from DB with the key.
- **Block Addition** – `Put()` adds a new block to the blockchain. The work flow of `Put()` is shown in Fig. 2. First it checks if the block is valid. If so, the block will be stored to DB and the blockchain will be updated. If not, it will return `success=false`. The rule of updating blockchain is as follows:
  - If `PrevHash` of the new block is the same as the blockchain's `LastHash`, it means the new block is on the current chain, then the `LastHash` will be updated and all the work will be continued on current chain.
  - If `PrevHash` of the new block is different from the blockchain's `LastHash`, and the height of new block is smaller than that of the blockchain's last block. It means the new block is on an alternative chain and the length of that chain is not longer than the current

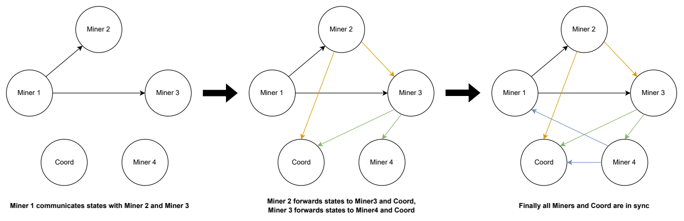
Miner 1
Miner 2
Miner 3
Miner 4
Coord
Miner 1 communicates states with Miner 2 and Miner 3
Miner 2 forwards states to Miner3 and Coord, Miner 3 forwards states to Miner4 and Coord
Finally all Miners and Coord are in sync

Figure 3. Gossip Protocol

chain. Then the `LastHash` will not be updated and all the work will be continued on current chain.
- If `PrevHash` of the new block is different from the blockchain's `LastHash`, and the height of new block is greater than that of the blockchain's last block. It means the alternative chain is longer than the current chain. Then the current chain will switch to the alternative chain. It iterates from the old chain and the new chain respectively, and finds the first different block. All the transactions from the first diffrent block to the end of the old chain will be collected to `oldTxns`, and all the transactions from the different block to the end of the new chain will be collected to `newTxns`. Finally, the `LastHash` will be updated to the hash of the new block.

3.3 Gossip

We implement the Gossip Protocol (Fig. 3) for information exchange between miners and the coord, including transaction broadcasting and block broadcasting. When a pending transaction is just submitted, only the miner that the client connects to has the information about the transaction. For other miners to learn about it, the miner needs to gossip about it with their neighbours until the pending transaction is fully propagated through the network. Similarly, when a new block is just mined by a miner, only the owner knows about the new block. The owner needs to gossip about it with its neighbours so that all nodes in the network can learn about the new block.

Since both the coord and miners need to participate in gossiping, we implement the protocol as a library so that it can be easily incorporated into the main logic of miner or coord. We implement two types of the Gossip Protocol for our system: *Push-Pull* and *Pull*. The *Push-Pull* strategy will be used by miners and the *Pull* strategy will be used by the coord, as shown in Fig. 4.

3.3.1 Push-Pull. Whenever miner notifies the gossip library through `UpdateChan` of a new block the miner just mined or a new transaction the miner just received from a client, a new push cycle will be triggered. At the start of each push cycle, the library will randomly select *FAN_OUT* number of peers as neighbours and connect to them. Then

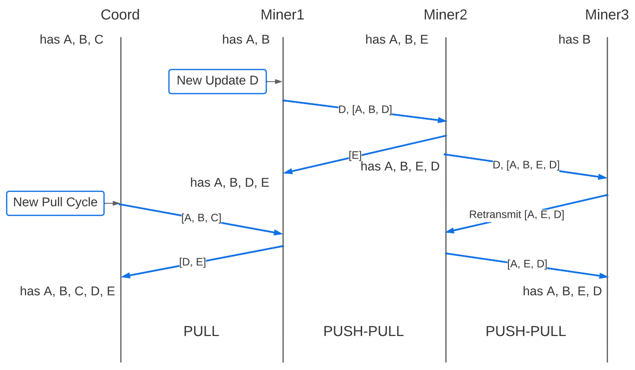

Figure 4. Illustration of Push-Pull and Pull

the library will push the latest update to along with its update history to the neighbours. When neighbours receive a push through, they will first check if they are missing any previous updates. If they are not missing anything, then they will (1) accept the update, (2) start a new push cycle to gossip about the update, (3) check if the pusher misses any update and reply with missing updates. Otherwise, they will not accept the update and request the pusher to re-transmit all missing updates. When the pusher receive the reply, it will add pulled updates first and then re-transmit updates that are requested by neighbours.

3.3.2 Pull. A new pull cycle is triggered at set intervals. Similar to *Push-Pull*, *FAN_OUT* number of peers will be randomly selected as neighbours at the start of a pull cycle. The puller will send its update history to neighbours and neighbours will check if the puller is missing any update. Then neighbours will reply with missing updates for the puller.

3.3.3 Failure Detection with Gossip. We also use the Gossip Protocol for miners to detect peer failures. When they push to a peer and the peer fails to respond, then they will remove the peer from its peer list and will not push to it in the future.

3.4 BlockVote

3.4.1 System Initialization. We use the coord to bootstrap the system and allow miners to join. The coord serves four purposes:

- Generate the genesis block
- Generate candidate list
- Track all active miners in the system
- Participate in block broadcasting via the gossip protocol

3.4.2 Miner Join Process. When a new miner wants to join the system, it first contacts coord to download a copy of the blockchain, candidate list and miner list by calling `Download`. Then it randomly selects a miner from the list and

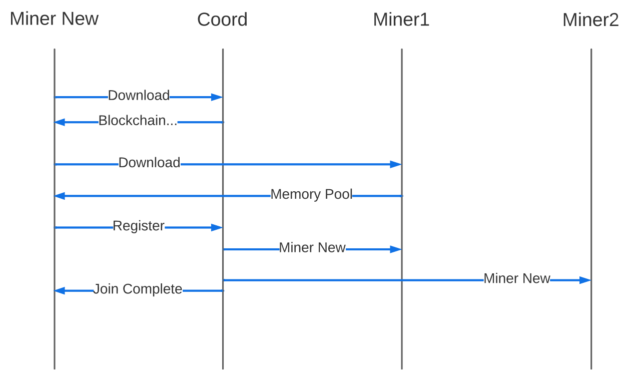

Figure 5. Miner Join Protocol

invokes GetTxnPool on the miner to download its transaction pool. After that, it registers itself as a miner at coord through Register. The coord will write the new miner's information to disk first before updating its view of the system. Then the coord notifies existing miners of the new miner by NotifyPeerList, and replies with a list of addresses for the new miner to gossip with. After the new miner receives the reply, the join process is complete. The join process is illustrated in Fig. 5.

3.4.3 Miner Failure Handling. Coord, miners and clients use different methods to detect miner failures. For coord, it uses fcheck to capture miner failures. When the coord detects a miner failure, it will first write to disk before updating its view of the system. Coord does not notify existing miners of the detected failure. However, when new miners or clients request for miner list, they will get the latest list from the coord.

For miners, they will use the gossip protocol to detect peer failures as described in Section 3.3. When miners detects a peer failure, they remove it from their gossip peers. Miners do not communicate detected failures with each other.

For clients, when they attempt to submit transactions to a miner but fail, they treat the miner as failed and remove it from their local cache of miner list and contact another miner in the list. If the miner list is empty, they will ask the coord to provide the latest list of miners and re-attempt.

3.4.4 Coord Failure Handling. Coord may fail and will restart after failure. To be able to recover from coord failure, coord writes to disk before updating its view of the system, including the blockchain, candidates and miner list. After recovery, it reloads them from disk and notifies existing miners of its new gossip address.

During coord failure, existing miners are unaffected and existing clients' transaction submission are also unaffected if no miner failure happens at the same time. However, new miners and clients will block until the coord recovers. Existing clients' query operations will also block.

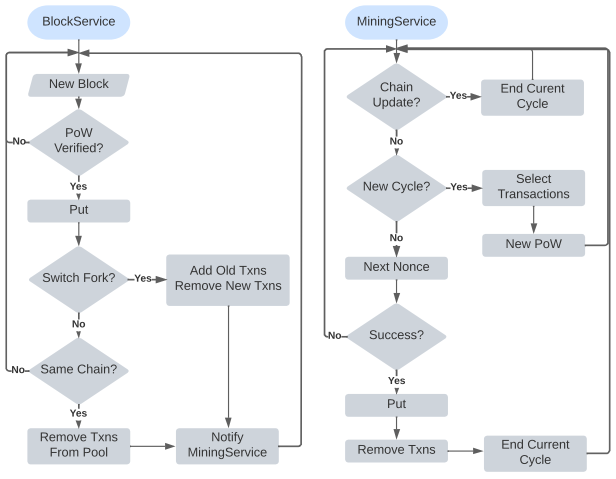

Figure 6. Miner work flow

3.4.5 Miner Logic. We implement the miner as four routines: main routine, TxnService, BlockService and MiningService. We show the workflow of BlockService and MiningService in Fig. 6.

Main routine is responsible for receiving block and transaction updates from gossip and dispatch them to TxnService or BlockService.

TxnService is responsible for processing transactions from clients or peers. It checks if a transaction has been received before, and append it to the end of the memory pool if it is a new transaction.

BlockService is responsible for processing blocks from peers. It will first check if the peer has completed the proof-of-work by calling the Validate() function on the block. Then it attempts to put the new block into the blockchain by calling the Put() API. If the API returns success, it will check whether the blockchain switches forks.

1. If no fork switching and the last hash of the blockchain is updated, then the new block is added to the current chain. All transactions included in the new block will be removed from the memory pool and the MiningService will be notified to restart the mining process.
2. If no fork switching and the last hash of the blockchain remains the same, then the new block is added to an alternative chain which is shorter than the current chain and therefore the miner can just ignore it.
3. If fork switched, then it will prepend transactions on the old chain to the memory pool and remove transactions on the new chain from the memory pool. It will also notifies the MiningService to restart the mining process.

MiningService is responsible for mining new blocks. At the start of each mining cycle, it will select *MAX_TXN* earliest transactions from the memory pool based on the time of

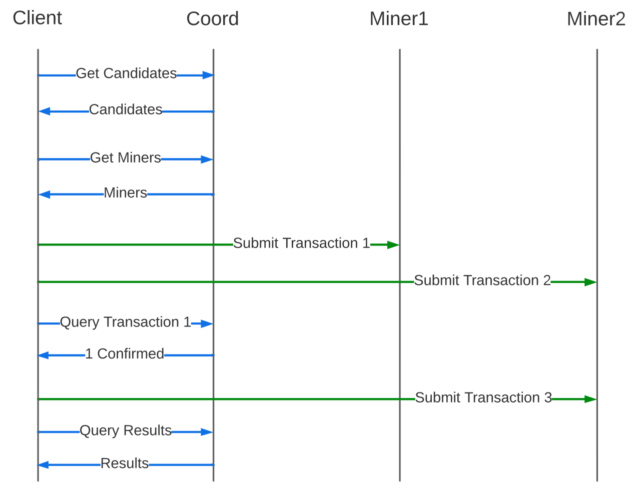

Figure 7. Client-coord and client-miner interaction

arrival and validate them in a batch. Selected transactions will not be removed from the memory pool at this point. If any transaction is invalid, it will be discarded. Then the miner creates a new proof-of-work puzzle and starts solving it. If the `BlockService` notifies it of changes in last hash, the current mining cycle will be termintated and a new cycle will begin. If the mining cycle runs to completion, which means it successfully mines a new block, it will add it to its copy of the blockchain, gossip about it and remove all transactions included in the block from the memory pool.

3.5 EVlib

We implement the evlib as an entry point for clients to access the system and interact with the coord and miners, as shwon in Fig. 7.

3.5.1 APIs. We implement four main APIs for clients: `Start`, `Vote`, `GetBallotStatus` and `GetCandVotes`.

- `Start(tracer *tracing.Tracer, clientId uint coordIPPort string) error` – starts an instance of EV to use for connecting to the system with the given coord's IP:port. If there is an issue contacting the coord, the method will always retry to connect until it retrieves an available miner address list. If there is an issue contacting the miner, the method will contact coord to get an updated miner address list.
- `Vote(ballot blockChain.Ballot) byte[])` – the client passes a ballot, and if the evlib does not store the information of this voter, it creates a wallet for the voter based on the voter's name and student ID. Then the method creates the transaction based on the ballot and wallet of this voter, Once the transaction is created, `Vote` will invoke `SubmitTxn` on a randomly selected miner in the miner address list to submit the transaction and returns the transaction ID for client to further query the transaction status.
- `GetBallotStatus(TxID []byte) (int, error)`– invokes `QueryTxn` on coord to retrieve the status of transaction, returns `NUMConfirmed` which is the number of blocks that has confirmed the transaction or -1 indicating the transaction hasn't been added to the blockchain.
- `GetCandVotes(candidate string) (uint, error)` – invokes `QueryResults` on coord to retrieve the number of votes a candidate has.

3.5.2 Client-coord Interaction. When clients start, they will contact coord to for candidate list and miner list before they can serve voters. Clients will not contact coord again for miner list unless all miners it knows failed. To check transaction or voting status, clients will send a query to coord, and the coord will use its local copy of the blockchain to return the result.

3.5.3 Client-miner Interaction. When submitting a transaction, the client randomly selects a miner and connects to it. If the selected miner fails to respond, the client will remove the miner from its miner list and attempt to contact next randomly selected miner.

3.5.4 Background Re-submission Routine. In the event of miner failure, a submitted transaction may be lost if the miner fails right after receiving it. We use a go routine to detect such lost transactions and resubmit them. When a new transaction is initially submitted in the `Vote()` API, we log the submit time and cache a copy of the transaction. The routine will periodically check if the time difference between now and the submit time exceeds *THRESH*. Then it will query coord to see if the transaction has been confirmed. If the coord cannot locate it in the blockchain, we resubmit it again and wait *THRESH* time.

4 Evaluation

To facilitate testing, we create multiple testing scripts:

- **Miner script**: starts given number of miners as separate processes, kills miners or start more miners upon request
- **Client script**: starts given number of clients as separate processes
- **Checker script**: checks whether the system is running properly based on the output files of all clients and the coord

Based on the properties of the system described in Section 2.5 and the voting rules described in Section 2.3, our checker script will perform the following four tests:

1. All **valid transactions** are committed **exactly once**.
2. All **invalid transactions** will **not** be committed.

3. Each group of **conflicting transactions** has **exactly one** transaction committed.
4. Vote **counts** are correct.

The checker script will operate on the output files generated by the system. When the system terminates, coord and clients will output multiple files, which represent coord's point of view and clients' point of view repectively. The coord will output all transactions committed in the blockchain and the number of votes for each candidate. Clients will output three types of transactions separately: valid transactions, invalid transactions and conflicting transactions (they don't know which one will be committed and therefore these transactions cannot be categorized into valid or invalid). The checker script is essentially checking whether the coord's point of view matches clients' point of view.

To fully test our system, we will conduct testing with the scenarios listed in Table 1.

Table 1. Evaluation scenarios of BlockVote.

	Coord Scenarios	Miner Scenarios
1	no failure	1 miner, no failure
2	no failure	multiple miners, no failure
3	no failure	multiple miners, up to n-1 failures
4	fail and recover	multiple miners, no Failure
5	fail and recover	multiple miners, up to n-1 failures when coord is alive
6	fail and recover	multiple miners, up to n-1 failures at any time

For all evaluation scenarios, there will be multiple clients each spawning more than 100 randomized transactions, including valid, invalid and conflicting ones.

In addition, during testing we add around 1.25 second delay in gossip to prevent propagation being too fast which significantly reduce the chance of fork switching. We also turn down the mining difficulty to 8 and add 50 milliseconds delay between each nonce to make the mining speed roughly the same across machines with different performance.

5 Demo Plan

First, we define the following notions:

- **Normal Operation:** The system processes transactions and updates the blockchain. To be more specific, clients submit transactions to miners, miners validate transactions and put them in the blockchain.
- **Survive:** When miners fail, remaining miners continue to process transactions and update the blockchain. When the coord fails, existing miners continue to process transactions and existing clients continue to submit transactions if there is no miner failure. After the coord recovers, new miners are able to join.

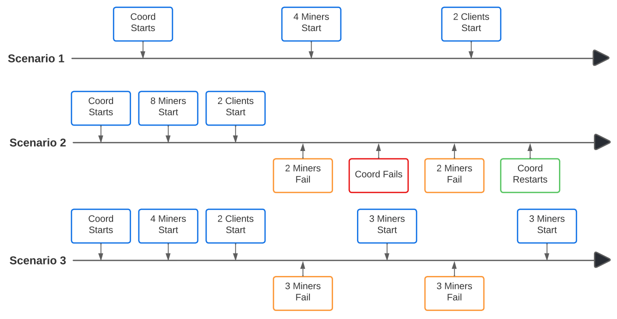

Figure 8. Scenarios for demo

- **Utilize:** Newly joined miners are able to validate transactions, mine blocks and participate in gossiping.

We plan to demo our projects under three scenarios described in Fig. 8, which will demonstrate normal operation, failure survival and new node utilization of our system respectively. Then we will run checker script on generated files to verify that the constraints are met in each scenario.

6 Conclusion

We design and implement a blockchain-based distributed digital voting system, namely BlockVote. Our system are comprised of the coord which bootstraps and coordinates the system, miners which validate transaction and store them to the blockchain, clients which serve voters by wrapping ballots in transactions. Our system provides the property that a transaction will be committed iff it is valid. Our system can withstand up to N - 1 miner failures, and can continue to work after the coord fails and recovers.

For future work, our system can be improved by having miners accept client queries and having the coord accept transaction submission so that our system can remain available as long as the coord or one miner is alive. We can also make client states persistent on disk so that client can fail and recover.

References

[1] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. 1987. Epidemic algorithms for replicated database maintenance. In *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*. 1–12.

[2] Johannes Göbel, Holger Paul Keeler, Anthony E Krzesinski, and Peter G Taylor. 2016. Bitcoin blockchain dynamics: The selfish-mine strategy in the presence of propagation delay. *Performance Evaluation* 104 (2016), 23–41.

[3] Satoshi Nakamoto. 2008. Bitcoin: A peer-to-peer electronic cash system. *Decentralized Business Review* (2008), 21260.