🅱witter Final Report

Group members: alika, adinkwok, ianmah, hong1999, huangsa, joshc822

Github repository:
https://github.students.cs.ubc.ca/CPSC416-2021W-T2/cpsc416_proj_joshc822_alika_ianmah_adinkwok_hong1999_huangsa

The Problem and Our Project

Twitter is a powerful social media platform, but there are growing concerns about the centralised control that its owners and/or governing bodies have over the platform.

🅱witter is a peer-to-peer version of Twitter backed by a custom blockchain network that is roughly based on Bitcoin/Ethereum to allow self-governance and increasing transparency versus a centralised network. The 🅱witter network has two main features: users posting and viewing tweets of all users within the system.

Design Decisions

Given the nature of Twitter's platform involving timelines, we required strong ordering semantics, specifically total ordering guarantees. In addition, a typical P2P system may suffer weak immutability guarantees because malicious attackers may try to modify the data they store. To enable these guarantees, we decided to implement our system with a blockchain that has characteristics that support the various distributed systems abstractions we desire. Blockchain provides immutability (achieved by the cryptographic hashes linking the chain), account verifiability, and total ordering (achieved by the proof-of-work/consensus algorithm inherent to blockchain).

Our Implementation

There are 3 types of components in our blockchain system: a coordinator, at least one or more mining nodes, and zero or more clients. The miner nodes make up a random graph of connectivity with a minimum of ***K*** neighbours following gossip protocol. The value of ***K*** is to be set in a configuration file for Coord. Clients interact with a mining node to post and view tweets.

Our implementation is heavily inspired by previous projects of this course and will be tested/deployed on the Azure Cloud.
https://www.cs.ubc.ca/~bestchai/teaching/cs416_2017w2/project1/index.html
https://www.cs.ubc.ca/~bestchai/teaching/cs416_2018w1/project1/index.html

Definitions

Coord/Coordinator: A centralised server to bootstrap the blockchain network of 🅱witter.
Miner/Node/Mining node: A node in the blockchain network.
Client: An external entity that interacts with a miner.
Mining pool/Network: The collective group of miner nodes that is managed by Coord.
Peer/Neighbour: A miner's neighbouring miner node assigned by Coord.
Joined: When a miner has been assigned peers.
Tweeth: Credits necessary for clients to create tweets and awarded to miners after successful mining.

Coordinator [Coord]

Our system uses a single coordinator node that connects nodes upon entry to bootstrap the system. The two main responsibilities of Coord are to inform miners of other miners in the miner pool and monitor them by sending heartbeats. **We assume that Coord never fails for our system.**

1. When a miner seeks to join the mining pool, it requests peer addresses from Coord. Coord returns ***K*** addresses (or all addresses if there are less than ***K*** other addresses in the pool).
2. Once a miner is ready to join the pool, it will notify Coord. Coord will add it to the mining pool and begin monitoring it with fcheck.
3. When a node fails fcheck, Coord will remove it from the mining pool and it

Miner

A miner node has the responsibility of mining blocks, receiving operation requests from the clients that are connected to them, and propagating operation requests/blocks to its peers. We incentivize mining with Tweeth, which is necessary for clients to create new tweets on the platform.

Miner Graph

Coord facilitates the creation of the miner network following as a random graph. Our initial design of the system leveraged unidirectional relationships between miner nodes. We realised that this decision inherited faults as joining nodes were able to send propagations but did not receive any. This occurred when all of the existing nodes in the network met their threshold of ***K*** addresses. Our solution to this issue was to make node relationships bidirectional. This allows new nodes to join such a network and have their ***K*** number of peers propagate to it.

Join/Rejoin Protocol

A joining mining node obtains the current blockchain from another node in the mining pool. Miner nodes first request ***K*** number of nodes from Coord with GetPeers(). It then tells Coord that it is available to receive blocks and queues any receiving blocks from its peers.

To receive the entire existing chain from a peer, a joining miner starts listening on a port to receive a file transfer of a blockchain file that is stored on the peer's disk (see details in Blockchain file storage section). The joining node makes an RPC request to the peer with its listening port and the peer initiates a file transfer. Once received, the joining node scans through the file and loads each block into memory, and validates the block. We wanted to avoid reading the entire file into memory as it is unknown how large a blockchain file may be. If the entire chain is valid, a ledger is generated for the block with the largest sequence number (indicating the longest chain), and the node starts its mining and propagation processes as two separate goroutines as we wanted a miner node to continuously perform these actions.

In our system, a rejoin is the same as a join. We previously considered these two protocols to be different, in which a rejoin would only start validation from the last known hash stored on its chain file from disk. However, this increased complexity when attempting to validate a chain that contained forks and was an unnecessary premature optimization. Since our system is currently not focused on scalability, we chose to have a rejoining miner continue to validate the entire file, including forks, because we want to verify the integrity of the entire chain as a fork may end up being the longest chain.

Mining

Miners add blocks to the blockchain by mining. A block contains the following:

- Sequence number [0….N]
- Previous (block in the chain) block hash
- List of posts/operations
- Public key of miner
- Nonce
- Current block hash
- Timestamp

Mining refers to a miner attempting to hash the contents of the block to meet a certain difficulty target. For example, we may require that the hashed result ends in 000. Miners achieve this by using trial and error and randomly generating nonces to achieve the desired hash result. Once a suitable hashed result has been found, this block is considered as "mined", or "found".

When a miner has successfully found a block, it records its current list of operations (that has until that moment been continuously updated with new operations) in the newly-mined block. It also takes the wallet data structure from its tail block, deducts the appropriate funds according to the list of operations, and then records the updated wallet in a map that has the block hash as the key, and the list of balances for each wallet as the value. It also writes the newly mined block to the existing blockchain and sends the block to the network. Nodes receiving the block will validate the block (see "Validation" section), then compare the list of operations recorded in the block with its own set of operations. Upon comparison, it will discard the operations that are included in the newly-attached block.

Each successful mine rewards the miner with Tweeth. For no-op blocks (blocks containing no operations), the reward is 1 Tweeth. For blocks with operations, the reward is 1 Tweeth + 1(numOperations) Tweeth. For example, if a miner is able to attach a block to the chain with 3 operations, it will receive 4 = 1 + 1(3) Tweeth as a reward.

Serving clients

- RPC function: error ← Post(PostArgs, PostResponse)
  - PostArgs: MessageContents (string), Timestamp (string), PublicKeyString (string), PublicKey (*rsa.PublicKey), SignedOperation ([]byte)

This is the RPC function that a client would call to make a post. All posts cost a constant amount of Tweeth (for now, let's say that each post costs 1 Tweeth). Upon receiving this request, the miner checks whether there is sufficient balance to perform the operation by comparing it to the wallet balance at the last mined block. If the miner has 0 Tweeth available, the RPC function should return an *InsufficientFundsError*. It checks the validity (described later, regarding RSA signature) of the operation. If the operation is valid, the miner subtracts the cost of the funds from a temporary copy of the latest version of the wallet and proceeds to propagate the operation request to all other miner nodes using our flooding protocol. The timestamp serves as a unique identifier, and can be used by nodes to prevent duplicates. When a block containing the requested operation is successfully mined, the opHash, blockHash, and tweethRemaining should be returned to the client.

- RPC function: error ← GetTweets(GetTweetsArgs, GetTweetsResponse)
  - GetTweetstResponse: BlockStack ([][]string)

This is the function that a client can call to view tweets. The miner tracks back through its blockchain and sends the client a stack of blocks. Using a stack allows us to push blocks as we backtrack through the chain and make it easy for the client to read blocks and consequently, tweets, in the correct chronological order. An important note for this feature is that we disregard the latest *n* blocks (in our implementation currently, *n* = 5). This means that the miner will wait for 5 confirmations of any block (5 subsequent blocks) before fully trusting it as a valid block.

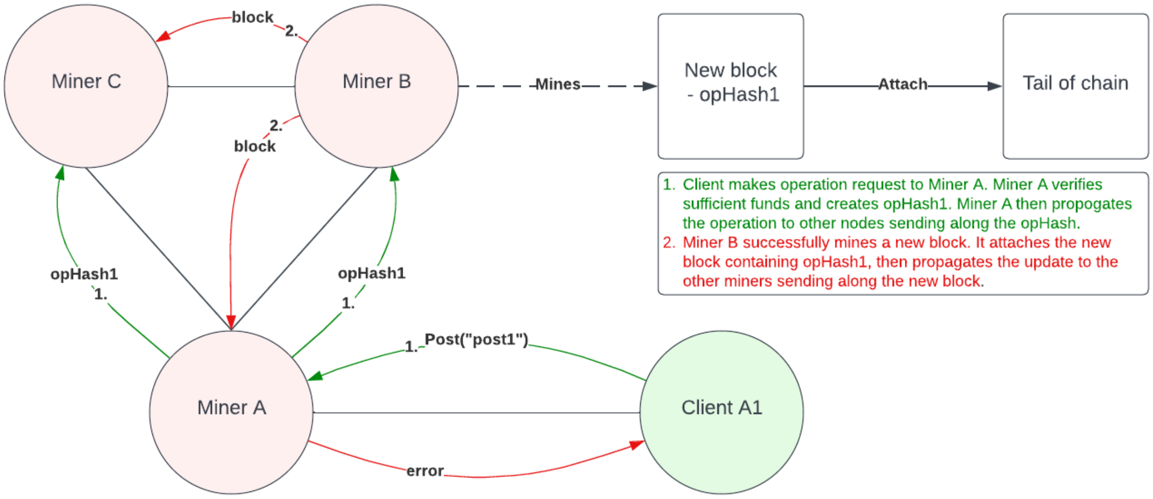
Propagation Protocol

Miner nodes are responsible for flooding other nodes in the mining network with two kinds of state: tweet operations, and blocks. Depending on the structure of the network:

- Once a miner receives a "Post" operation, it propagates this operation to neighbouring nodes.
- Once a miner mines a block, it propagates this block (its data, hash) to neighbouring nodes.
- A node propagates state to all neighbouring nodes and those nodes will do the same. It will stop relaying if the state has already been propagated.

If a miner node fails to propagate an operation to its neighbouring node after ***C*** tries, it deems that peer as failed. If this leads to a miner node having less than ***K*** neighbours, it requests more miner node addresses from Coord.

A challenge we faced during this implementation was when joining miners would miss propagations while validating the blockchain. To combat this, our implementation adds all incoming propagations to a queue. After the blockchain has been validated, it continuously works off of the queue.



1. Client makes operation request to Miner A. Miner A verifies sufficient funds and creates opHash1. Miner A then propogates the operation to other nodes sending along the opHash.
2. Miner B successfully mines a new block. It attaches the new block containing opHash1, then propagates the update to the other miners sending along the new block.

Blockchain File Storage and Forking

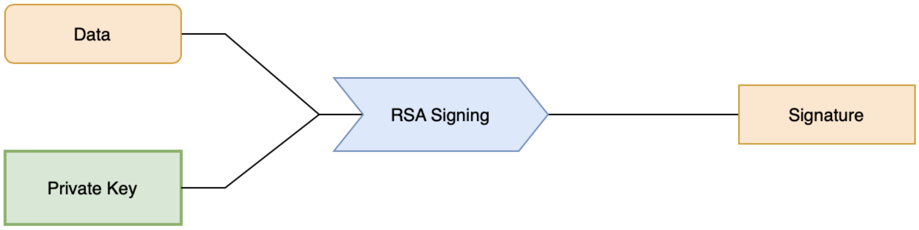
Since miners are independently mining blocks, there exists a race condition that is inherent to blockchain systems. This can happen when two nodes finish mining a block at the same time and propagate it to the network. Assuming both blocks (known as ommer blocks) are valid, a fork is now created and we record that within the chain storage file. Miners will keep mining and switch to the longest chain whenever one is determined. Miner's will ignore any block whose SeqNum is lower than their current Sequence Number minus 10. This is to avoid continuously building forks as well as to prevent malicious nodes from attempting to rewrite the entire chain.

The chain storage file is a simple text file that contains all blocks that have been mined and propagated on the network. Forks in the chain are represented blocks that may have duplicated sequence numbers but are linked by different previous hashes. We decided on this representation of the chain on storage because we noticed that there is no need for an elaborate structure to represent the forks/graph structure of the chain (e.g. json or nested arrays). The miner node network is primarily concerned with the longest chain, indicated by the largest sequence number, and is handled in memory before being written to disk. All nodes in the network will have chain files on storage that will eventually be consistent.

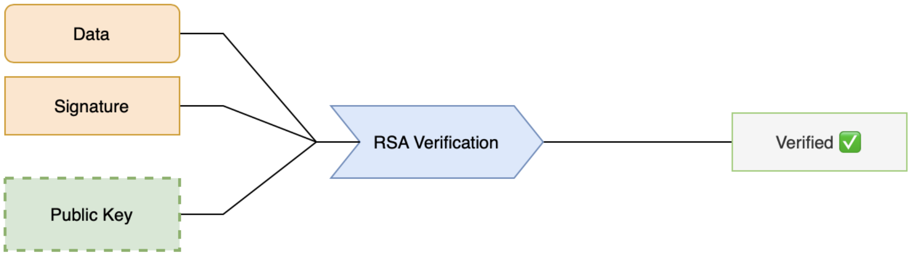
Validation

1. *Receiving a new operation from a connected client Source 1, Source 2, Diagram Source*

   A client sends new operations to any miner using an RPC function. We use RSA signing, using some miner's private key that is manually provided to the client when the client starts up. The miner receiving the operation uses RSA verification to verify the operation data matches the signature and public key that the client provides. This serves as a protection, only an actor who has the private key will be allowed to submit valid operations on behalf of that public address. It also verifies that the public key has enough funds to complete the operation (comparison last mined block), and then attempts to add the operation to a block. At this point the miner propagates the operation to its neighbouring nodes.



Signing of an operation



Verification of an operation

2. *Receiving a new operation from a neighbouring miner*
   When a miner is sent a new operation from one of its neighbouring miners, it must check the validity of the operation concerning available funds. The operation contains a public key. The receiving miner should look at the tail block of its chain and check the wallet of the miner associated with the given public key to validate that that miner has the necessary funds for the operation. It uses the combination of public key and timestamp and signature to protect against possible duplicate operations.

3. *Receiving a new block from a neighbouring miner.*
   When a miner is passed a new block from one of its neighbouring miners, it must check the validity of the new block. This means going through the operations in the block, and using the balances of the previous block making sure that each operation can be done using prior account balances. The miner must also check that the hash generated matches the list of operations + nonce to make sure the proof of work was not falsified. If a block has been determined valid, it is appended to the chain at seqNum - 1 (because sometimes we can receive an older block or two miners can finish a block at the same time causing a fork). We then check if the seqNum is the largest we have seen so far and update the current chain we are working to be that one if it is.

Ledger

A key feature of our system is the ledger. A miner will keep track of all posts/transactions through its ledger. Each propagated post and block is validated against the ledger. Our ledger data structure is map[string](map[string]int). As a high-level description, it maps ledger 'snapshots' to block hashes - each new block generates a new version of the ledger reflecting the transactions associated with the block. A ledger 'snapshot' (map[string]int) maps the public keys of miners to their respective wallet balances. Altogether we get a versioned ledger, where each version gives us an image of the state of the wallet after the miner receives a block.

Failure Protocol

Miner's Perspective
After ***C*** attempts to propagate a request to a peer, the propagating node deems its peer as failed. It removes that peer from its list of peers and relies on its other peers to maintain connectivity to the network (propagate operations, receive blocks, etc). If a miner *M*1 deems its peer *M2* as failed, but *M2* is passing fcheck from the Coord's perspective, *M2* will remain in Coord's mining pool, but *M1* will no longer have *M2* in its peer list. If the number of peers it now has becomes lower than some threshold, it requests new peers from Coord.

Coord's Perspective
If a miner fails fcheck with Coord, Coord deems that miner as failed. It removes that miner from the mining pool and no longer assigns it as a peer when nodes request neighbours.

Client

Instances of clients can run our library, bweethlib, to interact with the network. There are two main functions of the library: Post() and GetTweets(). This calls a miner's corresponding RPC calls. After the miner RPC call GetTweets returns, the library processes the returned BlockStack and prints existing tweets in chronological order.

Limitations

Eventual consistency for a social media platform

Eventual consistency for a blockchain system conflicts with some real-time properties desired in a social media platform. One main limitation for clients arises: the network is unable to notify them of when their tweet shows on the blockchain – potentially only providing an estimate at best for when a block with its transaction may be mined; even then, most blockchain systems wait after *n* blocks are mined before confirming that a transaction is valid because of forks.

Mining Monopoly

If malicious actors garner over 51% of the system's total mining power, they can potentially trigger a sybil attack. Attackers will gain control of all posts on the chain as well as the ability to omit incoming posts.

Coord Single Point of Failure

In the real world, Coord's integrity is an impossible guarantee. Coord acting as our bootstrap for the system makes it a single point of failure. The entire Bwitter network will not be able to operate if it experiences failures.

Mining Difficulty Selection

Mining difficulty directly affected our system's occurrences of ommer blocks. If the mining difficulty is set too low, we would see multiple forks within our chain, if the mining difficulty is too high, then having tweets appear on the chain would take a massive amount of time and could be deemed unreliable seeing those messages.

Evaluation

We evaluated our system's implementation based on the requirements outlined in this proposal by evaluating its behaviour under various scenarios throughout each of the milestones. We relied on logging locally to trace certain actions and states for the scenarios below:

Correctly runs through the happy path:
Join protocol:

- Miner nodes enter join protocol on start, Miner nodes should contact the Coord to get neighbours, Miner nodes download and verify the chain (unless genesis block), Miner makes join request, Coord adds miner to mining pool, Coord begins to monitor the miner with fcheck,
- **Evaluation**: chain.txt files are identical across nodes and follow total ordering

Mining:

- Miner continuously generates random nonces, Miner adds successful block to balance, Miner validates operations with public key, Miner enters propagation protocol
- **Evaluation**: chain.txt files are identical across nodes and follow total ordering

Post operations:

- Client issues tweet via POST API, Client signs operations with miners private key, Miner returns *InsufficientFundsError* if it has no balance, Miner validates the operation and performs propagation protocol, Each successful post subtracts some Tweeth from its balance
- **Evaluation**: chain.txt files are identical across nodes after a client post, wallet has been updated to reflect Tweeth cost

Propagation protocol:

- Miners propagate tweet operations to neighbouring nodes, Miners propagates mined blocks to its neighbouring nodes, Receiving miners verify incoming operation, Receiving miners verify incoming blocks, All nodes eventually have propagated state (block or operation)
- **Evaluation**: chain.txt files are identical across nodes, propagation will be seen through logs

Correctly handles bad path scenarios:
Node failures

- Coord, miners, and clients behave as expected as outlined in the Failure Protocol
- Evaluate by confirming behaviour after stopping the program of 3 nodes on Azure VM
    - Disk storage of blockchain information should be maintained
    - Each node will be removed individually and terminal logs of coord and affected nodes will be displayed to ensure the correct steps have been taken for failure recovery. Evaluation will be deemed successful if all peers who were previously connected to these nodes are shown to have been able to request new nodes from coord.

Node rejoining

- Miners and clients behave as expected as outlined in the Rejoin Protocol
- Evaluate by confirming behaviour after stopping the program of a node on its Azure VM and starting again after some time
    - Chain.txt files will be shown prior to the nodes rejoining and after rejoin has been completed, the chain.txt files from all nodes will be compared. Evaluation will be deemed successful if all nodes after rejoin have identical chain.txt files that display total ordering.

Blockchain forking

- Evaluate by hardcoding two miner nodes that finish mining at the same time and broadcasted. The block that gets validated across more nodes should become the main chain. The nodes with the other block will eventually continue work on the longer chain based on proof of work.

Malicious actor nodes

- Evaluate by hardcoding a malicious node to perform invalid block propagations
- Mining nodes should perform the correct validation steps as outlined Validation section, ignore the invalid block, and system should perform as expected in the happy path
    - Malicious blocks will be sent and the network will deny all of those blocks as they recognize the malicious messages. Evaluation will be deemed successful if chain.txt file after malicious nodes have been introduced remain identical across the non-malicious nodes, not including any malicious messages.

Malicious client (e.g. insufficient funds)

- Evaluate by hardcoding a client with a zero balance and make a tweet POST operation
- Mining node that receives the initial request should perform the correct validation steps as outlined Validation section, ignore the operation (i.e. do not propagate and return the correct error), and system should perform as expected in the happy path
    - Evaluation will be deemed successful if the chain.txt file of non-malicious nodes does not contain any of the malicious post transactions from the clients.

The system meets the following characteristics:

- Total ordering
    - After running the happy path and bad paths, we will evaluate that total ordering is maintained by logging the entire blockchain of all mining nodes and ensure that they are the exact same
- Immutability
    - We will evaluate immutability of the blockchain in the malicious actor nodes scenario. For example, a miner node that propagates a block that deletes an already posted operation/tweet
- Account verifiability
    - We will evaluate account verifiability of the blockchain by viewing the operations on the chain and tracing it back to the correct address

Out of scope:

- Scalability – we will not be collecting metrics on performance of the system with a large number of nodes and clients. We will have a maximum number of 5 mining nodes and 7 clients in our system (arbitrary numbers that will allow us to test distributed systems aspects; may be subject to change).

Demo Outline

The demonstration will be conducted through manipulation of our set up Azure nodes. Three miner nodes and coord will be set up before the demonstration time for the demo outlined below:

1. Join Protocol
    a. A new node will be introduced into the system outlining our systems join protocol. Chain.txt and terminal logs will be shown to demonstrate the process.
    b. Evaluation will be deemed successful if the newly introduced node is connected to the network and the chain.txt file is identical with existing nodes.
2. Mining
    a. Terminal logs will be displayed from a mining node outlining the mining process as outlined above showing blocks being sent to peers.
    b. Evaluation will be deemed successful when all appropriate steps of mining have been demonstrated alongside copious verbal explanation and terminal log walkthrough.
3. Post Operations
    a. A client will be connected to an existing mining node, the client will then send three tweets.
    b. Evaluation will be deemed successful if all three tweets have been accepted across the network. Chain.txt files will be compared across multiple nodes to show that the Post operations have completed and are a part of the chain.
4. Node Failures
    a. Each node will be removed individually and terminal logs of coord and affected nodes will be displayed to ensure the correct steps have been taken for failure recovery.
    b. Evaluation will be deemed successful if all peers who were previously connected to these nodes are shown to have been able to request new nodes from coord.
5. Node Rejoining
    a. To demonstrate our miner rejoin protocol, the removed nodes from the system that demonstrated miner failure will be restarted to join the network.
    b. Chain.txt files will be shown prior to the nodes rejoining and after rejoin has been completed, the chain.txt files from all nodes will be compared. Evaluation will be deemed successful if all nodes after rejoin have identical chain.txt files that display total ordering.
6. Malicious Nodes
    a. Malicious miner will be introduced that have been modified to accept all blocks from certain miners and will propagate modified blocks.
    b. Evaluation will be deemed successful if chain.txt file after malicious nodes have been introduced remain identical across the non-malicious nodes, not including any malicious messages.
7. Malicious Clients
    a. Malicious client will be introduced that are connected to modified malicious miners. These malicious miner nodes have been modified to ignore the wallet balance when any post transactions are made through that node.
    b. Evaluation will be deemed successful if the chain.txt file of non-malicious nodes does not contain any of the malicious post transactions from the clients.

Conclusion

🅱witter provides immutability, account verifiability, and total ordering backed by properties inherent to a blockchain system. However, its desired social media properties are also limited by eventual consistency.

Welcome to the 🅱witterverse.