# *int32coin: A Peer-to-Peer Cryptocurrency Protocol*

Jake Worden
Computer Science
California Polytechnic State University, San Luis Obispo
San Luis Obispo, United States of America
jmworden@calpoly.edu

***Abstract*—Implementation of a peer-to-peer decentralized blockchain, developed as a final project for an undergraduate distributed computing class. While this paper implements blockchain as a cryptocurrency tool, the technology has many applications as a general consensus protocol for decentralized networks requiring high Byzantine fault tolerance.**

**int32coin is written entirely in Go. Each int32coin client is run as multiple goroutines within a single process, and communicates changes within a peer-to-peer network to other client processes. The blockchain supports transactions between two wallets, and the sum of all incoming and outgoing transactions for a particular wallet defines that wallet's balance. A wallet is fundamentally a public/private key pair, and asymmetric encryption is used to authenticate transactions without needing centralization. Transactions are placed into candidate blocks before being added to the block chain.**

**Candidate blocks are propagated by peers to be mined, using a proof-of-work based blockchain validation algorithm to ensure that attackers can't fraudulently modifying transactions in the blockchain without expending large computational effort. Given a few optimizations suggested by previous work, and a relatively bug-free implementation, and attacker could only defeat the network by having more processing power than 50% of the network. Upon successfully mining a candidate block, the block is checked for validity against a strict list of protocol rules, ensuring the integrity the entire history and the integrity of each candidate transaction every time a new block is added. Only once the block is validated is it added and gossiped to peers, who will in turn attempt to validate the block. Coins are created as a mining reward for every newly added block.**

**The difficulty of mining reduces the frequency of divergent blockchains in the network, but doesn't eliminate them. In the somewhat rare case of conflicting history, the network will reach consensus by preferring the longest blockchain. This resolution procedure should be the primary focus of improvement, both in implementation and blockchain selection rules. Future work can also focus on peer selection, validation efficiency, generalization, and proof-of-stake.**

***Keywords—cryptocurrency, blockchain, consensus, peer-to-peer, cryptography***

## I. INTRODUCTION

Cryptocurrency, and the blockchain technology that support them, have many applications beyond the realm of decentralized currency. Indeed, all of the various blockchain offerings are fundamentally consensus protocols. Not only that, but consensus protocols with high Byzantine fault tolerance. Originally built to power a new type of monetary system, an ecosystem wherein attacks are not just possible but inevitable and frequent, blockchain had the benefit of being specifically designed for robust security without any obscurity from the beginning.

It certainly isn't the answer to every problem, but blockchain presents a novel solution to a complicated problem. It took so many recent developments in computer science just to make it work, and there's still so much left to improve.

### *A. Previous Work*

In [1], the paper ties together multiple key advancements in cryptography and distributed systems to define the world's first cryptocurrency protocol: *Bitcoin*. Asymmetric encryption is used to authenticate transactions without a centralized authority. Peer-to-peer networking is used to share transactions between clients. However, there was still a key problem that had stifled proponents of decentralized banking: double-spending.

If two transactions are broadcast to the network simultaneously, but the sender only has enough money to afford one transaction, how can the network guarantee that only one transaction is chosen? Bitcoin's solution was a proof-of-work

mechanism operating on hashes derived from the entire transaction history. In essence, a proof-of-work is value that is difficult to generate, but easy to verify.

Blocks, grouping of transactions, could only be added to the blockchain, the list of blocks forming the transaction history, if the block's hash is lower than some target value. By changing the block's nonce, a number that is persisted when the block is added to the blockchain but is otherwise meaningless, the block's hash will change. Mining, the processes of continually changing the nonce and recalculating the hash, is done until the block's hash is less than the target and the block can be added. While locating a valid nonce value can be computational expensive, validating the nonce is not.

Double-spending will rarely result in two simultaneous updates to the transaction history because the updates are effectively spread apart by computational cost. And even in the cases where it does, if the nodes in the network always prefer the longest blockchain (by number of transactions), one version will quickly win out since a block is mined over random intervals.

By including the hash of the previous block in the new block, a block is only valid if it originated from the same history. Changing a single transaction would only result in a valid blockchain if the changed block's nonce is recalculated along with every succeeding block's nonce. So proof-of-work also means an attacker could only alter the blockchain across the entire network if the attacking nodes have more processing power then all of the other nodes combined.

To incentivize mining, the node that finds the correct nonce is given the transaction fees specified by the senders along with a protocol-defined number of *Bitcoins* as a reward. This reward is the only mechanism to create *Bitcoins*. The reward decreases at fixed intervals, and the mining difficulty scales according to the speed at which the previous window of blocks were mined.

In [2], the Bitcoin protocol was adapted to create *Namecoin*, a decentralized peer-to-peer namespace and the first *altcoin (*a derived cryptocurrency). The domain-name mapping directory was encoded into blocks that would only be added to the blockchain provided a valid proof-of-work. This protects the registry from malicious changes without involving constant revalidation of entries. Miners are rewarded with *Namecoin*, as even when *Namecoin*'s purpose isn't to be a currency, it still relies on a large number of nodes to consistently mine new blocks. *Namecoin* also shipped with a generalized API that adapts the blockchain to store any generic name-value pair, facilitating the use of blockchain as a generalized consensus protocol.

In [3], the paper offers an optimization to *Bitcoin*'s method of resolving conflicting blockchains. There is a directly associated drawback with proof-of-work – it makes transactions take longer to be processed. Reducing the mining difficulty would make transaction turn-around time faster, but it would also increase the amount of blocks that are propagated across the network simultaneously. This will lead to more frequent divergent blockchains, commonly referred to as forks.

After the nodes in the network eventually reach consensus on the correct fork, the blocks in the fork left behind are now invalid and represent lost computational work. This is problematic if the blockchain protocol simply selects the longer chain, because an attacker trying to overtake the current blockchain can do so with a linear blockchain. The forks are irrelevant. So if the networks constantly is generating forks, it provides a computational advantage to the attacker's blockchain. The attacker can unilaterally prevent forks on its own blockchain and avoid unnecessary mining while the legitimate nodes waste cycles on blocks destined to be orphaned.

That's where the *Greedy Heaviest-Observed Sub-Tree* (GHOST) protocol comes in. Instead of picking the longest blockchain like *Bitcoin* does, *GHOST* will pick the heaviest blockchain. A blockchain is considered heaviest if the amount of computation that went into it's entire sub-tree structure, not just the main fork, is larger than any other sub-tree. This means the attacker would still need to expend the same amount of work as it took to create the main blockchain along with every orphaned block. Otherwise its fraudulent blockchain will never overtake the heavier legitimate blockchain.

In [4], the Ethereum protocol was introduced. It improves and optimizes many aspects of the *Bitcoin* protocol, including an implementation of GHOST so transactions can be processed faster without impacting security.
Another key optimization it brought about is *Sharding*, which is a mechanism to split the *Ethereum* network into subnetworks. This increases the scalability of the protocol by reducing that amount of validation checks that are constantly run to ensure blockchain integrity. As the blockchain lengthens, these checks require more and more computational resources. Eventually, the amount of computational power needed to keep up with the network would exceed the capacity offered by most consumer devices. This would implicitly incentivize centralization by making the network only usable on expensive commercial servers. *Sharding* counteracts this bias. Instead of every node in the network validating every transaction, nodes in *Shards* will only validate a small subset of transactions that are eventually merged into the main blockchain.

Not only did *Ethereum* optimize and improve on what *Bitcoin started*, but it also was packaged as more than a cryptocurrency. Continuing the generalization started by *Namecoin*, *Ethereum* was designed as an extensible distributed system platform. Along side its component cryptocurrency, *Ether*, *Ethereum* includes numerous libraries that isolate and generalize many features of its blockchain. Some of its libraries – namely, *Ethereum*'s cryptography packages – were used to implement this project. The platform has also been used to create everything from smart contracts to *CryptoKitties*, a game about collecting and trading cartoon cats [5].

### B. Objective

*int32coin* certainly isn't designed to compete with *Bitcoin* or *Ethereum*. It's a simplified implementation of a cryptocurrency developed as a learning tool. Among the previous work mentioned (and not mentioned), *int32coin* bares the most resemblance to *Bitcoin*. This is not because *Bitcoin* is the best cryptocurrency or was intentionally emulated, but because it is one of the simplest by nature of being first. So the average cryptocurrency prototype probably will look a lot like *Bitcoin*.

It would be interesting to expand *int32coin* as a more generalized blockchain consensus protocol rather than a cryptocurrency. It's current form was only selected because it would be straightforward to implement. There's plenty that can be achieved in a few weeks, but blockchain is such a huge field that touches on many different aspects of computing, so there's also so much more work to be done.

## II. Blockchain Implementation

The *int32coin* protocol was implemented entirely in Go. The blockchain module, encapsulating the local protocol of the cryptocurrency, is composed of five components.

TABLE I. Blockchain Module

| **Component** | ***Description*** | ***Sect.*** |
|---|---|---|
| Wallet | Client wallet, sends/receives transactions | A |
| Transaction | Coin exchange from sending wallet to receiving wallet | B |
| Block | Queued transactions awaiting addition to blockchain | C |
| Blockchain | List of validated blocks | D |
| Router | Routes internal messages between modules | n/a |

Fig. 1. Components that comprise the blockchain module, and the sections wherein they are individually discussed. The router component contains little logic, and thus is not given its own section.

The *Blockchain* and *Router* components run concurrently as goroutines.

### A. *Wallet*

The wallet is akin to a user account, and is used to send and receive transactions. The wallet is effectively a wrapper for a pair of public and private keys along with a derived identifier. These key pairs are generated using Elliptic Curve Cryptography (ECC). Specifically, they use the Secp256k1 curve, which is also used in the *Bitcoin and Ethereum* protocols [1,2].

The generated public key is 512 bits. To generate the address, the public key is hashed using sha3 (used throughout the *int32coin* protocol to hash values) to generate a 256 bit address. This wallet address is the only value used to identify a wallet, but it isn't actually guaranteed to be unique. However, due to the nature of ECC key generation and the size of the address, collisions are so improbable that they may as well be considered impossible.

Theoretically, should a collision occur, that wallet would effectively be identified by two addresses and thus could be signed using either of 2 associated private keys.

TABLE II. WALLET STRUCTURE

| **Field** | **Wallet Contents** |
|---|---|
| | ***Description*** |
| Private | Full ECC private key |
| Public | Full ECC public key |
| Address | Wallet address, derived from public key |
| Transactions | Unused |

Fig. 2. Contents of the wallet data structure.

As the name would imply, the private key should be kept private by the wallet owner. Knowledge of the private key constitutes ownership of the wallet, since it is used to sign transactions.

### B. *Transaction*

A transaction represents a transfer of coins from a sending wallet to a receiving wallet.

TABLE III. TRANSACTION STRUCTURE

| **Field** | **Transaction Contents** | |
|---|---|---|
| | ***Description*** | ***Bits*** |
| Sequence | Sequence number, unique to block | 32 |
| Sender | Sending wallet address | 256 |
| Receiver | Receiving wallet address | 256 |
| Amount | Integer number of *int32coin*s to be sent | 32 |
| Signature | Signature generated by private key of sender | 256 |
| TXID | Transaction ID, unique to transaction by sender | 256 |

Fig. 3. Contents of the transaction data structure.

The message digest is derived from the *Sender*, *Receiver, Amount,* and *TXID* fields. To generate the digest, each field is concatenated together and hashed, then the entire hash is hashed again.

Once the digest is generated, it can be signed with the private key of the sending address.

The transaction ID is randomly generated for each new transaction, which will uniquely identify a transaction by sending wallet address. The point of this ID is so that duplicate transactions can be detected by the protocol. By enforcing that the transaction ID is unique by sender, any single transaction is guaranteed to be included in the blockchain only once.

It wouldn't be unreasonable to view the transaction ID as globally unique. For the same reason that the wallet address is unique in practice, the transaction ID is also unique in practice by nature of its size and random generation. However, transaction IDs being locally unique to each sender is actually enforced by the protocol – if the transaction ID has appeared before with the same sending wallet

address, the transaction/outer candidate block will be rejected.

To validate the signature of a transaction, the public key used to sign the transaction can be extracted from the signature and digest. This key is hashed, and compared to the sender's address. If the two values match, the transaction is considered authentic.

### C. Block

A block is a grouping of transactions, and is used to compose the blockchain. Candidates are blocks that have not yet been added to the blockchain. Mining is a process, described below, to determine a value for *nonce* that will turn the candidate block into a valid block ready to be added to the blockchain.

TABLE IV. BLOCK STRUCTURE

| **Field** | **Block Contents** | |
|---|---|---|
| | ***Description*** | ***Bits*** |
| Height | Height of block in blockchain | 64 |
| Nonce | Value incremented by miners | 64 |
| Previous hash | Hash of previous block | 256 |
| Merkle root | Root of Merkle Tree containing each transaction in this block | 256 |
| Target | Value that this block's hash must be less than | 256 |
| Transactions | List of transactions in this block | n/a |

Fig. 4. Contents of the block data structure.

The Merkle root is calculated by generating a Merkle tree of all the transactions (hashing the entire transaction, not just the digest). Each transaction is hashed twice. The hash within the root node of this Merkle tree is then stored in the block. Changing a single transaction in the block will completely change the Merkle root.

The actual hash of the block is not stored in the block itself, since it needs to be computed each time it's needed to verify the integrity of the block. To generate the digest, the *height*, *nonce*, *previous hash*, *Merkle root*, and *target* fields of the block are each hashed and concatenated together. Then, as with transactions, the entire hash is hashed again. This means that changing any one of those fields will completely change the resulting block. Collisions, of course, are still possible – the security of the blockchain is dependent of the strength of the sha3-256 hash function.

The *nonce* and *target* fields are used by miners to demonstrate proof-of-work. To mine a block, the miner will first add a *reward* transaction to the start of the block's transaction list. The specifics of the reward are statically defined by the protocol, so if the reward does not match the specification of the blockchain component, it will fail verification. Once the reward has been added, the miner will increment the nonce and calculate the block's hash. This hash is compared to the target. The block is only considered successfully mined if the hash is less than the target. The "difficulty" of mining is inversely proportional to the size of the target. If a miner finds a nonce value that produces a hash smaller than the target, the miner will request the block be added to the blockchain.

The final value for nonce is the miner's proof-of-work. It is difficult to generate, but easy to verify.

### D. Blockchain

The actual blockchain component is the core of the *int32coin* protocol. It links all the local components together, and enforces the rules of the cryptocurrency.

TABLE V. BLOCKCHAIN STRUCTURE

| **Field** | **Blockchain Contents** |
|---|---|
| | ***Description*** |
| Height | Height of blockchain |
| Blocks | Ordered list of added blocks |
| Queue | Queued transactions awaiting addition to |

| Field | Blockchain Contents |
|---|---|
| | ***Description*** |
| | blockchain |

Fig. 5. Contents of the blockchain data structure.

Candidates are checked according to the following rules:

- The block hash must be less than the target hash.
- The block must have the correct target hash.
- The block's previous hash must equal the hash of the top block on the blockchain.
- The calculated Merkle root of the block must equal the included Merkle root of the block
- The block height must be one higher than the blockchain height.
- Each transaction in the block must be properly signed by the sending wallet.
- Each transaction in the block must have sufficient balance in the sending wallet.
- Each transaction in the block must have a transaction ID that has never been used by a transaction with the same sending wallet.
- The first transaction in the block must be the reward to the miner.
- The miner must be awarded the correct number of coins.
- The reward must have been sent by the root wallet (which can only be used to reward miners).
- The block cannot contain 0 transactions

Should a candidate violate any one of these rules, it is rejected and not added to the blockchain. Otherwise, the block is added, and all transactions in the queue are removed and requeued. If the new queue is not empty, it is used to populate a new candidate block to be sent to the miner and distribution module.

Whether the transaction is enqueued automatically or via interactive input from the command line, it is always verified first according to the block validation rules enumerated above (only the ones that can be applied to a single transaction).

A blockchain is always newly initialized with a genesis block. This block can contain any arbitrary transactions and is not subject to the normal protocol rules. However, the block is still hashed and all succeeding blocks will need to be attached to the genesis block – if two blockchains have differing genesis blocks, then they are not capable of being merged.

## III. Distributing the Blockchain

Each instance of the blockchain module, packaged with the distribution module, is run as a single process. In additional to the cryptocurrency rules, the *int32coin* protocol also describes the peer-to-peer network all nodes are a part of.

The distribution module implements this network, and serves to communicate changes in the blockchain to peer *int32coin* nodes. This includes candidate blocks to be mined and validated blocks to be added. Transactions are not shared unless added to a candidate/validated block, though they could be.

### *A. Networking Backbone*

This implementation uses *libp2p*, a peer-to-peer library built on TCP and written in Go, to communicate with other nodes. The primary functionality drawn from this package involves creating addresses for peers that can be used to easily establish connections so long as the target address is known. The *libp2p* library implements

far more then just addressing – it includes many builtin distributed system protocols – but they are left purposefully ignored since it would defeat the educational purpose of this project. To determine what could be utilized, this heuristic was used: "Would manually implementing this feature be interesting? Or would it only be tiresome?"

In the case of the TCP backbone and node addressing, tiresome seemed a more fitting description.

### *B. Distribution Protocol*

The root of the distribution component is the *manager* goroutine, which orchestrates the communication with peer nodes and acts as an interface between the network and the blockchain module.

To establish a connection to a peer, that peer is dialed. After dialing, both the peers launch a goroutine to handle the connection: one for reading, and one for writing. Each goroutine will route messages to/from the *manager* goroutine.

Messages between peers are sent in three ways:

- Gossiped to a random subset of connected peers.
- Broadcast to all connected peers.
- Unicast to a specific peer.

The blockchain is distributed as follows:

- When the blockchain generates a candidate block, it is broadcast to every peer for mining. The decision to broadcast this message instead of gossip it was somewhat arbitrary.
- When a node receives a candidate block from a peer, the block will be sent to the miner component for mining, canceling the current mining job.
- When the blockchain validates and adds a new block, it is gossiped to peers.
- When a node receives a validated block from a peer, it will validate and add the new block to its blockchain.

### *C. Peer Discovery*

To join the network, a node has to pick an entry point – the first peer it wants to connect to. To do this, it can either dial an address specified via command line argument, or randomly select one from a file. While this connection may fail because the target node is down/malfunctioning, a node will never reject a peer request. So even if a node chooses not to connect to an address it is given by a peer (detailed below), it will still allow anyone to connect to it. As long as there is an active entry point one has knowledge of, there is a way to join the network.

When a node establishes a connection to a peer, it initiates a *hello* handshake/exchange. The two newly connected nodes trade information about their blockchain and their peers. If either node has a shorter blockchain, it will request a bulk transfer of the remaining blocks from the new node.

After that process is completed/skipped, the node will compare its newly connected peer's list of peers to its own list. It marks each address as seen, and it collects all the addresses that it isn't connected to. After marking each address as seen, it gossips the list of just-seen addresses to all of its peers.

Then, it inspects all the addresses it could potentially connect to. If it already has connected to a sufficient number of peers (a parameterized goal), it will continue without adding any peers. Otherwise, it will attempt to fill its remaining peering slots, dialing each address until there are no more slots. The order that it attempts to fill slots is random.

Improvements to peer discovery would include an algorithm to randomly disconnect peers and request a new list of peers from neighboring peers, allowing the network to be connected more randomly.

### *D. Failure Detection*

Failure detection is primarily handled by TCP or libp2p. When the distribution module detects that a connection has been closed or timed out, it tears down all the resources associated with that connection and purges the address from any local data structures (including the list of seen addresses).

## IV. Results

Testing was not straightforward, as it's not a trivial processes to generate test data due to the blockchain validating every new transaction and block. To overcome this difficulty, each node was given a random transaction generation module, which would create mostly valid transactions between multiple newly created wallets. It could then package these transactions into a candidate block and broadcast it to the network. By setting the mining difficulty high (set by environment variable), and calling the testing module through a rudimentary command line interface, a large amount of network traffic could be generated (since en-queued transactions automatically are broadcast as a candidate block should a new block be added).

### *A. Blockchain Integrity*

The transaction validation protocol appears robust, and at no point in testing did a node's blockchain reach in an invalid state. Regardless of block origin – whether it be a bulk blockchain transfer, a local miner's internally transferred solution, or an externally validated block shared by a peer – the block would always be verified according to the protocol's rules before being committed to the blockchain. Even internal modules are not viewed as trustworthy sources by the verification process, and the protocol does not provide exceptions to the verification step. Most of this protection is implicitly enforced by providing a single interface to add a block.

There is a performance cost to diligent verification. In particular, checking each transaction's sending wallet for sufficient balance is particularly inefficient. The check for a unique transaction ID occurs at the same time. In the current implementation, a single balance verification involves inspecting every transaction in the entire blockchain. This check is applied for every new transaction, so the cost is compounded. As the blockchain gets to a nontrivial length, the individual transaction inspections are a significant bottleneck to the network. This issue was the primary motivation for *Ethereum's Sharding [4]*. There are many minor optimizations that could be made to improve performance, such as aggregating transactions in a single block by sender (as is done by Bitcoin [1]) so that a sender's balance isn't unnecessarily recomputed. However, the easiest way to substantially reduce the cost of verifying new blocks would be to better index the transactions. Generating a transaction database, which could be indexed by sending wallet address and transaction ID, would bring about large speed improvements without necessitating a design overhaul. This transaction database could be separated from the blockchain data structure itself, only storing valid transactions already committed to the blockchain.

The other checks to guarantee blockchain integrity are comparatively trivial. As such, running the per-transaction checks last allows obviously invalid blocks to be rejected quickly. This is particularly beneficial given the amount of redundant blocks that are gossiped by the network. By validating block height and hash values first, a node is able to quickly discard blocks that it has either already seen or originate from an incompatible history.

### *B. Byzantine Fault Tolerance*

For reasons mentioned above, the blockchain has strong protections against Byzantine faults. Poisoned history, forged transactions, and duplicate transactions were all quickly identified and rejected by the blockchain module.

The peer-to-peer network, however, is nowhere near as secure. There are a number of ways a malicious node could attack peer nodes. From general network attacks such as slow lorris or denial of service, to exploiting the distribution protocol. Should anyone want to use this cryptocurrency for practical purposes, improving network security to Byzantine faults is by far the most pressing issue.

When a node joins the network, it will accept the first blockchain it receives (which could be sent by an attacking node). While the blockchain will be rejected if it fails verification, this doesn't protect it from accepting an independently valid blockchain that isn't the "actual" blockchain. The protocol's protections are based on what the node believes to be the current blockchain – upon joining, the node only knows about the genesis block. And if a block disagrees with it's peers about the contents of the blockchain, it is not going to be able to accept blocks from its peers. If it disagrees with all peers in the network, then that node is effectively nonfunctional.

This means that even if the blockchain protocol is extremely secure, the only thing backing the protections is consensus among the distributed network – which has an open admittance policy. If an attacker is able to trivially take down nodes, which one currently could, it would be easy to take control of the network and essentially invalidate any of the builtin protections.

### *C. Race Conditions*

This implementation sees many goroutines spawned per node, and an arbitrary number of nodes in the network sending multiple types of messages that need to be processed differently, so the elimination of race conditions is certainty nontrivial. Within the time-frame of this project, it was not feasible to eliminate all of them. So, during triage, only the race conditions that would bring down the entire network were viewed as critical. Random node failures are certainly inconvenient for the owner of the node, but they only have a minor impact on the overall network (mainly involving the work done by neighboring nodes to properly close the connection). As long as the failure of a peer can't propagate through the network, the protocol is operationally unaffected by the existing race conditions. Of course, it would be better if they were eliminated, and some were.

One noteworthy race condition involved connecting to a peer. The connection needs to be established on both ends, but the logic for the node that initiated the connection is different from the node accepting the connection. Additionally, there is a limbo period after establishment wherein the receiving node doesn't know any information about the newly connected node, and has to wait for the other node to start the *hello* handshake. This process leaves room for two nodes to both initiate a connection to one another within a short time-span, which are always accepted by the other node. Left unchecked, the duplicate connections would waste local resources, generate needless network traffic, and could lead to unforeseen complications.

This was solved by simply closing a connection should one of the nodes learn it is redundant. To ensure that only one of the connections is closed, and only closed once, only the target node with the lexicographical larger address will close the connection. The nature of the connection establishment process guarantees that both nodes agree on which connection came first, so

synchronization only requires explicit selection of the node to initiate the teardown. The node with the smaller address will simply ignore the redundant connection (but still keep it open in separate goroutines), only performing local teardown after initiated by the corresponding peer.

There are other deficiencies that were left unresolved. One of which being the bulk blockchain transfer (initiated during the *hello* exchange if one of the nodes has a shorter blockchain). This portion of the protocol leaves much to be desired, and the process is not reliable. The blockchain of the peers that have already synced with the network are not effected, so the network as a whole is still reliable. And since the synchronization process is still subject to verification, a failed transfer still results in a legal blockchain. However, from the joining node's perspective, synchronization failing in the middle of block exchange is less than ideal. Currently, the only available recourse to the node would be to leave and rejoin the network (thus restarting the synchronization procedure).

The root cause of this issue could be stem from multiple protocol rules. For starters, the blockchain height is considered during every *hello* exchange – meaning that a node that starts off by connecting to multiple peers will initiate and process many blockchain transfers concurrently. At the very least, this causes unnecessary network congestion. The issue could also arise from how the transfer takes place: block-by-block and without acknowledgment. Should one of the peer's messages be lost, the peer will still continue sending the succeeding blocks – all of which will be deemed invalid by the receiving node because their ancestor block is nonexistent.

A possible fix to this issue would be to add an acknowledgment mechanism, and only start one blockchain synchronization at a time (by selecting the longest blockchain and not requesting an exchange until the active one has completed/timed out).

### *D. Divergent Blockchain History*

Since the fundamental purpose of blockchain protocols are to establish consensus, divergent history resolution is an important performance metric for any implementation. Considering the *int32coin* protocol is decentralized and peer roles are homogeneous, the best the network could do is guarantee eventual consensus. However, this guarantee is only partially implemented. Divergent histories are resolved during blockchain synchronization, which is only called during the *hello* handshake. This procedure is capable of resolving partial divergence – discarding the blocks that disagree with the remote blockchain and beginning synchronization from the highest similar block.

The protocol doesn't specify when synchronization should occur outside of connection establishment, so the only way that a node in this implementation could resolve history would be to reconnect to the network and submit to the peer's blockchain over it's own. This is problematic because the node should be capable of resolving blockchain forks without connecting to anyone new.

It also brings into question the permanency of the blockchain, since transactions would be undone and need to be reapplied. Or they could even be "lost" by the network. Loss could be prevented by repackaging the orphaned transactions into new candidate blocks, but some transactions are bound to become invalid because of insufficient balance. A transaction being orphaned would also appear to both sender and receiver as the network canceling a transaction, which could be concerning or misunderstood by an end user.

Bitcoin represents the risk of orphaning blocks to users as transaction confirmations. Each transaction has a confirmation count, with the first confirmation resulting from adding the block to the blockchain,  and subsequent confirmations resulting from blocks being added on top of the original block [1]. Providing transaction confirmations effectively leaves deciding when a transaction can be considered permanently added to the blockchain up to the owner of the receiving wallet. Each confirmation reduces the likelihood that the transaction's block is in a divergent blockchain, but a new confirmation will only be added at lengthy intervals (once every average block mine time).

That said, the primary mechanism to resolve divergent blockchains is never to generate them in the first place. This is one of the main benefits of using proof-of-work based mining: the probability of two unique blocks with valid correct proofs-of-work being propagated by the network simultaneously decreases as the mining difficulty increases. *int32coin* was originally intended to dynamically scale the difficulty according to external conditions, though this was only partially implemented. Should the protocol be amended to support that, divergence can be permanently suppressed (but never completely avoided).

## V. Conclusion and Future Work

As is mentioned by almost every section within the Results chapter, there's still plenty of room for improvement. Many of them are just fixing bugs with the current protocol (e.g. blockchain synchronization), or fully implementing features that have been started (e.g. dynamically scaling difficulty and miner reward, automatically resolving divergent blockchains).

There's also protocol features that can be expanded, such as improving peer discovery and selection, or increasing byzantine fault tolerance on the networking side.

Past improving what's already there, so much is still left to explore. In particular, it could be worthwhile to modify *int32coin* with mining based on proof-of-stake instead of proof-of-work, as is done in *Ethereum [4]*.

All that being said, I think the project made it pretty far considering the how much time it was allotted.

## References


[1] *S. Nakamoto, "Bitcoin: a peer-to-peer electronic cash system," 2008.*

[2] H. Kalodner, M. Carlsten, P. Ellenborgen, J. Bonneau, and A. Narayanan, "An empirical study of Namecoin and lessons for decentralized namespace design," WEIS, 2011.

[3] Y. Sompolinsky and A. Zohar, "Secure high-rate transaction processing in Bitcoin," 2015.

[4] G. Wood, "Ethereum: a secure decentralised generlised transaction ledger," 2014.

[5] CryptoKitties, "CryptoKitties: collectible and breedable cats empowered by blockchain technology, " 2018.