Proof-of-Work Blockchain in Go

David Mberingabo
Carnegie Mellon University

Parardha Kumar
Carnegie Mellon University

Abstract

Our project demonstrates a Proof of Work Blockchain network using Go. The network consists of registered worker nodes and users who can also register to the network. Each node and user is represented by a port number on the local host. This implementation does not use public/private key infrastructure, and as a result, it does not make any cryptographic guarantees. However, most of the logical mechanisms offered by Proof of Work are as described in the Bitcoin white paper [1].

1 Introduction

Proof of Work (PoW) is a consensus mechanism used in blockchain networks to validate transactions and maintain the integrity of the network. Satoshi Nakamoto originally introduced it as part of the Bitcoin protocol in 2008. In a PoW system, network participants, also known as "miners," compete to solve a complex mathematical puzzle that requires significant computational power. The first miner to solve the puzzle is rewarded with newly minted cryptocurrency and the right to add a new block of validated transactions to the blockchain. This block is then broadcasted to the network for other nodes to validate and add to their copy of the blockchain. The computational difficulty of the puzzle is adjusted over time to ensure that new blocks are added to the blockchain at a fixed rate, regardless of the number of miners participating in the network. This difficulty also helps to prevent the creation of fraudulent or malicious blocks that could harm the network. While PoW is known for its security and resilience to attacks, it also has drawbacks, such as high energy consumption and scalability limitations. As a result, alternative consensus mechanisms, such as Proof of Stake (PoS) and Delegated Proof of Stake (DPoS), have been developed to address these issues.

2 High Level Description

The network begins as two empty lists, one for nodes and another for users. Nodes and users have assigned ports that are unique to the list. These files are accessible to the public and new nodes/users can look up registered participants to register. These files are stored in the */tmp* folder in our Unix implementation. Once a blockchain has been created, registering nodes must get the current blockchain from 2/3rd of the network to join the network. The genesis block is auto-created once there is a non-trivial number of nodes on the list (in our implementation we chose 4 nodes to be the minimum). Anyone can request any node for the current state of the blockchain using our JSON API request */copy_chain*, which is how nodes can remain updated on the blockchain state.

If $2/3^{rd}$ of the nodes on the list are unreachable, then there can be no consensus, the blockchain cannot accept content and no other nodes can get added to the network. This flaw is because we do not implement a way of removing nodes from the network, should they crash. The network can only grow, so as nodes crash, it can become impossible to achieve majority consensus on anything as some of the nodes are unreachable but peers do not know. In our test cases, we never reach a point where $2/3^{rd}$ of the nodes are unreachable. A possible solution to our limitation is to vote on removing nodes that do not respond to a threshold number of requests, such as */copy_chain*, in order to have only active nodes participate in consensus.

Satoshi Nakamoto suggests that each node should be listening for users and users should send their requests to multiple nodes, to ensure content acceptance. To simulate this suggestion, our clients will broadcast their content to a set of randomly chosen network nodes and each of the nodes must do the following:

1. Block Mining: Collect new user content into a block and mine a PoW for the block. (Limitation: One content request per block)

2. Block Acceptance/Rejection: If no valid block has been

accepted from peers by the time the node finishes mining, the node broadcasts the block to all peers and waits for a majority of 2/3rd to accept the block, or reject it.

3. Block Validating: If a newer block is received while mining (or idle), the node should validate the received block. If it is valid, accept it, otherwise continue mining.

4. Validating Concurrent Blocks: Resolve blocks that are received while validating (i.e. concurrent blocks).

A block's Proof-of-Work should be replicable by all other nodes given a common hashing algorithm and the corresponding hash and nonce. Our implementation uses JSON-encoded blocks and SHA256 to mine for corresponding hashes and nonces. To keep blocks unique in our local demonstration, we added a timestamp field. Since Satoshi's design requires users to send content to multiple nodes, for a practical implementation users could provide some content signature that can be checked to not cause multiple requests when a user wanted only one to go through. A block is valid iff:

1. it has a valid index

2. its Proof-of-Work is valid

3. the block is not already in the chain (No double spend!) and

4. a valid index is higher than the most recently committed block index

A valid index is higher than the most recently committed block index. When a node receives a block that skipped an index or more, the node realizes it is missing a block and it should update its blockchain by asking its peers. This is a drawback from our demonstration, because instead of sending entire blockchains for updating, Satoshi suggests simply sending the missing blocks. We further acknowledge this limitation and others in our code comments and README.md.

Satoshi stresses that nodes always consider the longest chain to be the correct one and will keep working on extending it. When a different but valid block arrives while another is being validated, we say the two valid blocks arrived concurrently. Satoshi describes rules a node must follow when accepting blocks that arrive concurrently. For example, block B arrives concurrently while block A is being validated. When it's time for acceptance and block B is the same as block A, then the node should accept the earlier block A. If block B has a different index from block A, then the receiving node should choose the block with the larger index and update its blockchain.

If the concurrent block's indices are tied, then the earlier block should be accepted. Satoshi asks us to store the other block as a branch in case it grows larger, but we do not do so in our implementation. This is because our nodes are able to update their entire blockchain when they grow out of consensus, so what we lose in network efficiency, we gain in stability and ease of achieving consensus. Ties are simply broken by either the longer branch or the earlier one. With the tie broken, the other branches should be rejected. This relies on the simple rule that "Nodes always consider the longest chain to be the correct one and will keep working on extending it." Although this is a deviation from Satoshi's instructions, it is more of an efficiency limitation of our implementation than a theoretical one.

3 Design Requirements

Design requirements are specified in the form of this proposal, an API and test cases. All these can be found in our GitHub repository [2]. We will test these requirements: Registration: This requirement will test how new users and nodes become registered, and how a new blockchain is created. Block Mining and Acceptance: These requirements will test the proof of work process, as well as the broadcasting and acceptance process. Consensus and Conflict Resolution: The conflict resolution process will also be tested by sending concurrent blocks and testing how nodes resolve these conflicts and come to consensus over time. Conflicts will be demonstrated by an array of validated blocks, whenever a node is about to accept a block, it first checks if there are any conflicts by checking whether the array of validated blocks only has one block or multiple. If there are multiple blocks, then only one must be chosen for acceptance, following Satoshi's rules for conflict resolution.

4 Programming Specifications

Our implementation will rely on common Go modules such as crypto/sha256 and encoding/JSON for cryptographic hashing of blocks during Proof of Work operations.

While our implementation checks for double spending, it does not use a Merkle Tree as suggested by Satoshi Nakamoto, which means our implementation would scale poorly. For demonstration purposes, a simple lookup of the blockchain for that hash will suffice.

Our design utilizes broadcasts to establish majority consensus, however, this would be extremely costly in a practical network. Our implementation does not address this scaling limitation, but the idea remains that consensus eventually propagates to all nodes in some P2P manner. Our way is a simple broadcast/response to all registered nodes. This is in order to avoid many of the complex optimizations required for a practical P2P network. Our API calls will rely on the net/http module

5 Test Case Specifications

Our test implementations can be found in our GitHub repo. [2] Our tests will follow our requirements. Nodes and users must be able to register to the public list of nodes and users. They must automatically choose ports that are not on the list already. The nodes must create a new blockchain only once four nodes are on the list. Once a blockchain is created, all registering nodes must request the latest committed blockchain from peers. Thread safety of the public lists is not tested and is considered a requirement for future considerations.

Block mining can be tested by acting as a user, asking a node to add content to the blockchain. Once a block is accepted, it must have updated the majority of peers' blockchains. This can be tested by sending the */copy_chain* request and verifying majority consensus. Here are different user scenarios that can be tested: A user can send the same content to multiple nodes, and only one instance of that content must be accepted. A user or many users can send different data to multiple nodes, and only one of the data content should get accepted to the blockchain at a time. Since a block's index is part of its PoW, a user must make a new request if the block containing their previous request does not get accepted for that index spot. A user receives a failure response for each rejected content.

6 Acknowledgements

Our sincere gratitude to Professor Patrick Tague, TAs Aditya Desai, Akshay Katheria, and Zeean Veljee for their support and guidance toward this study.

References

[1] Bitcoin: A peer-to-peer electronic cash system.

[2] Github.