# Introduction

The article [] raises the question of some issues of the standard Bitcoin protocol []: the problem that big amounts of computation resources and energy are wasted by miners that are not successful in appending a new block and the problem of high latency of transaction confirmation (high delay between the time when a transaction is proposed and the time when the transaction can be considered confirmed with high probability). To deal with those issues, the collaborative block reinforcement protocol is proposed. The main idea of this protocol is to use the proof of work (POW) to reinforce a new successor block, regardless of which miner found it. Instead of defining the longest chain as the valid one, the chain with the most associated amount of reinforcement is considered as valid (the heaviest one). This semester project is devoted to the practical implementation of the aforementioned protocol proposed in the article by using Python and asynchronous framework Twisted and making experiments which will confirm its efficiency (in terms of solving the aforementioned issues) in comparison with the standard Bitcoin protocol.

# Protocol description

The short description of some concepts of the protocol, which are used on this project, are provided below. As was mentioned in the article [], on the high level, the algorithms of the collaborative block reinforcement protocol are the following:

1. Miners do their work on top of what they believe is the last block of the blockchain.
2. During this mining, POW is produced at each miner locally.
3. When a miner ***m*** succeeds in computing a POW that is sufficient to append a new block, it makes a ***propose*** of a new block (broadcasts it).
4. When other miners see the proposal, they may decide to reinforce that block by sending their POWs to ***m***.
5. ***m***, after collecting the reinforcements from other miners, ***commits*** the block (broadcasts the reinforcements).
6. Miners start new mining on top of new block (It is only possible to mine on top of committed blocks).

Some of these stages need more precise explanation and it will be done below.

In order to make the POW independent from a new block, the mining process will not directly involve the new block. Miner choose what block to append after generating a POW on top of the last block. For the generation of the POW, miner tries to find a nonce ***n***, such that ***h(A, n, pub\_key) < d***, where ***h*** is a hash function, ***key*** is miner’s public key and ***d*** is the mining difficulty parameter, ***A*** is the last block. If miner does not find a nonce that would produce a hash < ***d***, it may save nonces that produce considerably small hashes. These nonces, although not sufficient for appending a new block, can be later used to reinforce a block found by another miner.

In order to keep the probability of forks small, it is necessary to guarantee that a POW is only used to reinforce one block. If both blocks B and B’ are successors of A, the miner, in order to maximize the chances to be rewarded, may have the incentive to reinforce both blocks (make double reinforcement). Such behavior must be prevented and for this purpose the concept of proof of misbehavior (POM) is used. Any pair of messages where miner tries to reinforce such two blocks with the same nonces is considered as the POM against that miner. Awarding reinforcement-related rewards are delayed for k blocks after the block they are associated with. The first miner that was able to append one of these k blocks and have a POM against malicious miner may claim the reward of malicious miner by adding a POM to the block.

The usage of the heaviest chain as the valid one requires the definition of ***weight***. The weight of the POW is:

where ***h*** is the hash corresponding to the nonce, ***d*** is the mining difficulty parameter. The weight of the block is the sum of the weights of all POW supporting this block (by definition, the weight cannot be less than 1). The weight of the chain is just the sum of the weights of all blocks which it consists of.

Taking into consideration that allowing reinforcements of any weight might flood the network with reinforcements of small weight, second parameter () is used that sets the bound on the weight of allowed reinforcements (in order to be used during reinforcements the following inequality must be satisfied for the nonce: hash < ).

# Event-driven programming and Twisted

Event-driven programming is a programming paradigm in which the flow of the program is determined by events. In an event-driven application, there is a main loop that listens for events and triggers a callback function when one of those events is detected. This model of programming perfectly suits the idea of the distributed system which is going to be implemented (and which must react to such events as network messages and finding of suitable nonces). There is a design pattern that has such behavior – reactor pattern – which is realized in the Twisted framework for Python. For this reason, this framework was chosen for the project.

# Implementation details

The implementation of the protocol consists of modules each of which has a particular purpose (blockchain, broadcast, hash, states, POM, setup). The client that produces transactions is implemented as a separate program. The parameters of the implementation are: timeouts of reinforcement collection and commit receiving, threshold for switching of mining, mining difficulty for appending a block and reinforcement, number of blocks during which the POM is looked (k in the protocol description) and some parameters which allows to change the behavior during experiments (add later).

# Blockchain

In the implementation of blockchain two types of blocks are used in order to satisfy the protocol: propose blocks and commit blocks. A propose block contain the following fields: a nonce, miner’s public key, the list of included transactions, a link to the block on top of which it was mined, a link to the corresponding commit block, a timestamp and a flag which shows if the block was added by a malicious miner and is used during the experiments (see the corresponding section in the report). A commit block contains the following fields: a link to the corresponding propose block, the list of links to the blocks mined on top of this, the proof of misbehavior (POM), all the reinforcements done by miners, the weight of the chain terminates with this block, a timestamp and a flag which shows if the block was added by a malicious miner. In order to simplify the sending of blocks over the network, the ability to serialize them into the JSON format and the ability to create blocks from JSON format is added.

The implementation of blockchain allows the following main operations: addition of the block of each type and returning the block which is the last one in the heaviest chain. The storage of blocks in the blockchain is organized by levels each of which corresponds to the depth of block. For instance, first (genesis) block has depth equal to 0, next block has depth equal to 1, etc.). It is done in order to simplify the addition of a new block which uses the depth and the hash of the previous block to find that previous block. During the addition the validity of blocks in terms if the nonces produce small enough hashes is checked. In order to make the experiments, there is the ability to return the heaviest block only in the chain which terminates with block in which the malicious flag is set. In order to deal with blocks which arrive not in the proper order because of the network issues, the pool of blocks is used. If the previous block for arrived block is not in the blockchain, arrived block is added to this pool of blocks and is retrieved from it when the previous block is added to the blockchain.

# States

In order to make the implemented program well-structured and easy to debug, the protocol logic is programmed in the form of final-state machine with three states: the ***mining*** state, the ***reinforcement collecting*** state and the ***reinforcement*** (was) ***sent*** state.

When the mining on top of the last block of the blockchain starts, the final-state machine is in the mining state. In this state the following events are processed:

1. Nonces, found by hash module (see the description of the corresponding module). If the hash is sufficient for appending a block to blockchain (less than mining difficulty parameter for appending mentioned above), hashing module is stopped, a new propose block is created, appended to a local copy of blockchain and broadcasted to all miners. State is switched to the ***reinforcement collecting*** state. If the hash is not sufficient, the nonce is stored and the mining continues.
2. Propose messages, delivered by broadcast module (see the description of the corresponding module). The propose blocks based on these messages are created and added to the local copy of blockchain. If the propose block was mined on top of the block on top of which current mining is going, current mining is stopped by stopping the hash module and all stored nonces are broadcasted as a reinforcement message. State is switched to the ***reinforcement sent*** state.
3. Commit messages, delivered by broadcast module. The commit blocks based on these messages are created and added to the local copy of blockchain. Messages are also transferred to the POM module (see the description of the corresponding module). If the difference between the weight of the chain which terminates with the added block and the weight of the chain which terminates with the block on top of which current mining is going is greater than or equal to the threshold for the switching of mining (mentioned above as the implementation parameter), current mining is stopped by stopping the hash module and the new one on top of the last block of the blockchain starts (in such case there is no switching of state). It is done in order to avoid continuing mining when there is a heavier chain which is very likely to become the valid one.
4. Reinforcement messages, delivered by broadcast module. They are transferred to the POM module.

In the reinforcement sent state the following events are processed:

1. Propose messages, delivered by broadcast module. The propose blocks based on these messages are created and added to the local copy of blockchain.
2. Commit messages, delivered by broadcast module. The commit blocks based on these messages are created and added to the local copy of blockchain. Messages are also transferred to the POM module. If the commit which was received is the commit for the propose which initiated the switching from mining state to the reinforcement sent state, the state is switched to ***mining*** state and the new mining on top of the last block of the blockchain starts. To avoid stacking in reinforcement sent state due to network conditions or the malicious behavior of any miner which sends propose messages and never sends corresponding commit messages, there is the timeout of commit receiving (mentioned above as the implementation parameter). After this timeout expires, the state is switched to ***mining*** state and the new mining on top of the last block of the blockchain starts.
3. Reinforcement messages, delivered by broadcast module. They are transferred to the POM module.

In the reinforcement collecting state the following events are processed:

1. Propose messages, delivered by broadcast module. The propose blocks based on these messages are created and added to the local copy of blockchain.
2. Commit messages, delivered by broadcast module. The commit blocks based on these messages are created and added to the local copy of blockchain. Messages are also transferred to the POM module.
3. Reinforcement messages, delivered by broadcast module. All nonces in the reinforcement message are checked if they produce small enough hashes. Those nonces which are valid in this term are stored. In order to restrict the time during which the reinforcements are collected, there is the timeout of reinforcement collection (mentioned above as the implementation parameter). After this timeout expires, the commit blocks including all collected reinforcements and the POM (if there is one) is created, added to the local copy of blockchain and broadcasted to all miners. Miner’s own nonces which are stored during the mining state as significant enough to reinforce but not significant enough to append a new block are also included as reinforcements. The reinforcement messages are also transferred to the POM module. The state is switched to ***mining*** state and the new mining on top of the last block of the blockchain starts.

In order to compare the collaborative block reinforcement protocol and the standard Bitcoin protocol, the latter was also implemented. To simplify the implementation and to reuse the existing code, it was based on the blockchain implementation for the collaborative block reinforcement protocol (with the pair of propose and commit blocks (the latter does not contain any reinforcement) considered as a block) and it also uses the concept of state (however, it has only one state which never switches). In this state the following events are processed:

1. Nonces, found by hash module. If the hash is sufficient for appending a block to blockchain (less than mining difficulty parameter for appending mentioned above), hashing module is stopped, new propose and commit blocks are created, appended to a local copy of blockchain and broadcasted to all miners. The new mining on top of the last block of the blockchain starts.
2. Block messages, delivered by broadcast module. The propose and commit blocks based on these messages are created and added to the local copy of blockchain. If the block was mined on top of the block on top of which current mining is going or if the difference between the weight of the chain which terminates with the added block and the weight of the chain which terminates with the block on top of which current mining is going is greater than the threshold for the switching of mining, current mining is stopped by stopping the hash module and the new mining on top of the last block of the blockchain starts.

In all the aforementioned states the messages containing transactions from clients are processed. The transactions are added to the pool of transactions.

# Hash

This module produces nonces whose corresponding hash is less than mining difficulty parameter for reinforcement mentioned above. The hash is obtained by SHA256 hashing miner’s public key, nonce and the hash of block on top of which the current mining is going. The nonces are checked sequentially, starting with nonce equal to 0. When the appropriate nonce is found, the state module is informed. The search for nonces continues until hash module is explicitly stopped.

(are the difficulty parameters explained before?)

# Setup

This module generates the configuration files necessary to the correct behavior of the miners. These files are: one private file for each miner and client containing its private key, and one public file containing the miner’s and client’s public key, IP address, port and the configuration of the program (i.e. the timestamp of the genesis block, which miners are malicious and which version of the protocol should be run).

The public–private key pairs are generated using the *PyCryptodome* library.

# Broadcast

This module is responsible for the communication between the miners. It receives a message from a miner, signs it with the miner’s private key and sends it to all the other miners over a TCP connection. When the message is received, the signature is verified and, if valid, the message is delivered to be processed by the application, otherwise it is ignored.

The broadcast abstraction is implemented using the *zeroMQ* library which has bindings for the Twisted framework, facilitating the integration with the reactor. Out of the several message passing models that this library supports, we used the “Publish – Subscriber” one. In this pattern, each miner is both a publisher and a subscriber of all the other miners. As a publisher it sends its own messages and as a subscriber it listens and delivers the messages of all the other miners.

There is also a tag parameter offered by the library that is used to label the different types of messages (i.e. reinforcements, proposals, commits, …)

# Reinforcement POM

The Reinforcement POM module is responsible for the prevention of double reinforcements, as in [2.3 of paper].

Every time a reinforcement or a commit message is delivered all the reinforcements there contained are stored in a data structure. This structure is based on dictionaries to provide a fast lookup time. If any of the nonces included in a reinforcement was already used by the same miner to reinforce another block propose block associated with the same previous commit block a POM is found. The reward associated with these reinforcements can now be claimed by including the original messages in the next commit block that this node broadcasts.

To reduce the memory used by this structure, we use the fact that the blocks that are far from the head of the chain will unlikely be forked to delete the reinforcements used in those blocks.

By default no miner will double reinforce. So, to test our implementation we created a parameter “-f” that changes the behavior of the miner in order to always reinforce a new propose block with all the reinforcements found so far.

# Client

The client module simulates the clients of the blockchain. They continuously broadcast random transactions that will be delivered by the miners and included in the next commit block.

# Experiments

The goal of the project is to compare the performance of the Reinforcement with the Bitcoin Protocol. We focused our experiments in the likelihood of forking the longest chain. A chain harder to fork will lead to a reduction in the number of blocks needed to consider a block valid and reduce the latency of the transactions.

To perform this comparison, we implemented a protocol version similar to the one used in the Bitcoin system [article Bitcoin]. To reuse the blockchain data structure already implemented we maintained the propose and commit blocks. This pair of blocks is considered the “Bitcoin block” and is broadcast in one single message. When a node receives this “propose + commit” message it stores it in the blockchain and restarts mining on top of the longest chain, no reinforcements are sent.

In all the tests, there is a set of “malicious” and a set of “honest” miners. The “honest” ones mine always on top of the heaviest chain as it is the logical behavior to maximize their rewards. The “malicious” ones have the goal of forking the main chain many times they will not mine on top of the heaviest block.

Every node produces a log that describes all the messages sent and received as well as all the blocks appended to the blockchain and the block on top of which it is mining. So, further analysis to these logs can be done afterwards.

# Test conditions

To have runs as independent as possible we generated new keys and a new propose block before each run.

Each miner was a separate process in the same machine that communicated with the other nodes through TCP connections.

The machine used to run the tests used the operating system El Capitan 10.11.6 with a 2.2GHz quad-core Intel Core i7 processor.

There was a total of 7 nodes, 4 honest miners and 3 malicious ones.

The exact configuration constants used in the program are included in the appendix.

# Test cases

# Cancel every block

In this configuration, the malicious miners try to cancel each new block committed by the honest nodes.

Reinforcement version:

When the nodes are launched, they all start mining on top of the Genesis block. If a malicious miner finds the next block it advertises all the other malicious nodes via a separate channel (we used the broadcast module to simulate that separate channel), so the others can know that they should wait from a propose from that node and then reinforce it. This maximizes the utility of the malicious set. When an honest node proposes a new block, it will receive reinforcements from the other honest nodes and then commit. At this point, if the malicious set have already found a block they propose, reinforce and commit it. If they did not they give up.

So, at the end we will have appended to the genesis block either a propose and a commit block from the honest miners or a propose and a commit from both. If the malicious chain is the heaviest the malicious have successfully cancelled the other block. At this point all the nodes will restart mining on top of the heaviest chain and the malicious will try to cancel the next block, the block at depth 2.

Bitcoin version:

The principles are the same as the ones explained above except that the malicious miners can never be successful with only one block because, on the best case, the malicious and the honest chains will have both weight 1. To tiebreak, the winning chain is the one that finds the second block first. The possible outcomes of one iteration are:

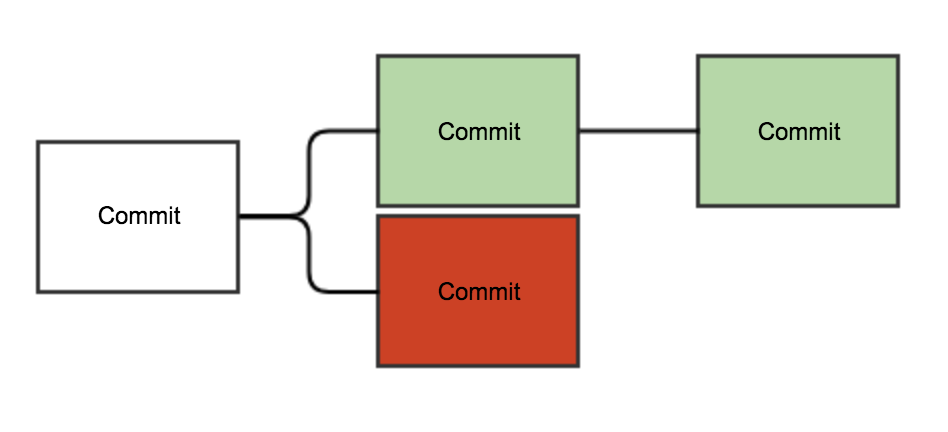
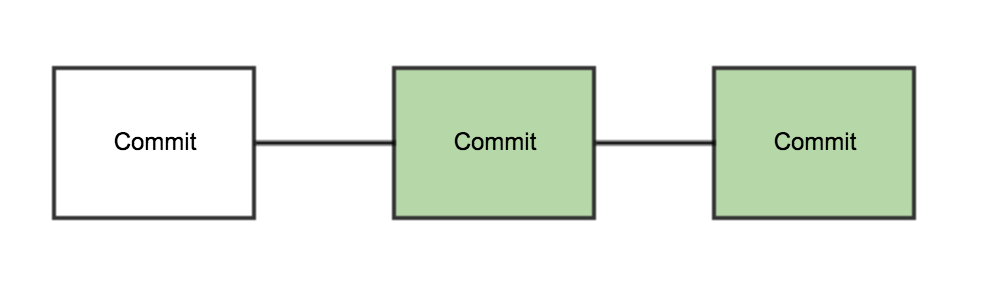
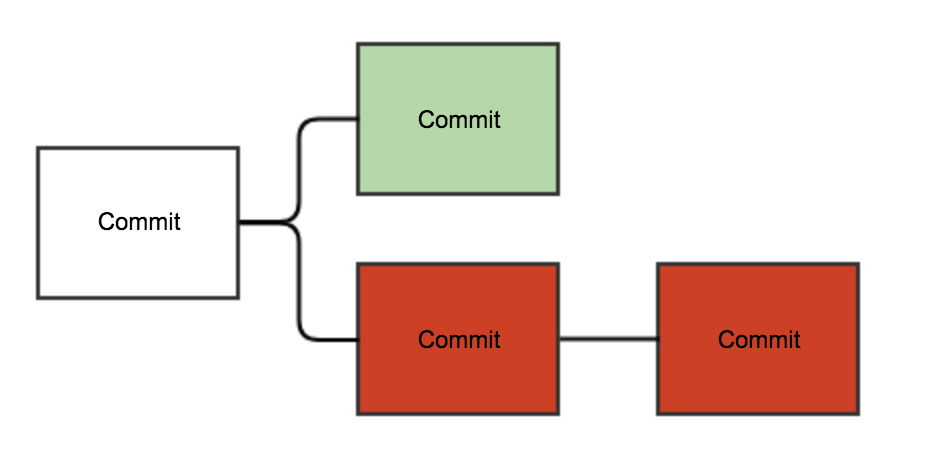
  

Figure 1: Outcomes of the Cancel Every Block strategy – Reinforcement Version

In the first two diagrams the winning chain the honest one, whilst in the last the winning chain is the malicious one. As before all the nodes restart mining on top of the longest chain and perform, again, the same actions.

# Results

We ran each of the versions 5 times. In each run the malicious miners had the opportunity to cancel a maximum of 25 blocks, which means that the Reinforcement Version run until the depth of 25 and the Bitcoin Version until the depth of 50.

|  |  |  |  |
| --- | --- | --- | --- |
| **Run** | **Honest wins** | **Malicious wins** | **% Honest wins** |
| 1 | 22 | 3 | 88% |
| 2 | 19 | 6 | 76% |
| 3 | 19 | 6 | 76% |
| 4 | 21 | 4 | 84% |
| 5 | 18 | 7 | 72% |

Table 1: Cancel Every Block – Reinforcement Version

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Run** | **Honest wins** | **Malicious found the 1st block** | **Malicious wins** | **% Honest wins** |
| 1 | 22 | 7 | 3 | 88% |
| 2 | 23 | 5 | 2 | 92% |
| 3 | 25 | 4 | 0 | 100% |
| 4 | 25 | 5 | 0 | 100% |
| 5 | 24 | 4 | 1 | 96% |

Table 2: Cancel Every Block – Bitcoin Version

# Analysis

The results are unexpected, the Bitcoin protocol performed better than the Reinforcement protocol. It was easier to find a competing block heavier than the block broadcasted first, than to find 2 two blocks before the honest majority found one.

One of the reasons that motivates this difference it the fact that no selfish mining strategy was implemented (i.e. in the Bitcoin protocol, when the malicious miners find a block before the honest miners they simply stop mining and wait for the “honest block”. If, instead of stopping, they started mining on top of the block they already found they would increases their chances of finding the second block first). This strategy would increase the chances of the malicious miners in the situation where they found the first block first (3rd column of the table) and would balance the results.

The Reinforcement Protocol is not affected by this strategy because, while the malicious nodes are waiting for the “honest block” they continue discovering hashes that will be used as reinforcements and thus increase the weight of the next “malicious block”.

# Cancel a Particular Block

In this configuration, the miners pre-agree on the depth, *d*, of the block they want to cancel (it is written in a public configuration file) and try to cancel it.

From depth 0 to depth *d*-2 only the honest miners try to append new blocks. At depth *d*-1 the malicious nodes try to find a new block and, if so, they advertise the others and wait for the “honest commit”. Then, the malicious propose a block, reinforce and commit it. Notice that now, if the malicious miners don’t find a block before the honest miners they do not give up, they keep trying until they fork the chain at the desired location. From this point on the malicious miners will only mine on top of the malicious chain, trying to make it as heavy as possible.

The winning chain is decided either when: the malicious chain becomes the heaviest one and the honest miners start mining on top of it, or the honest chain is much heavier than the malicious one (much heavier is a constant in our program and should be set to a value that is recoverable with very low probability).

We expect worse results in this test than in the cancel any block because it does not give up if it does not cancel the block immediately.

# Results

We ran each of the versions 20 times. The depth of the block to cancel was randomized between runs.

|  |  |  |  |
| --- | --- | --- | --- |
| **Runs** | **Honest wins** | **Malicious wins** | **Malicious wins with only 1 block** |
| 20 | 1 | 19 | 10 |

Table 3: Cancel a Particular Block – Reinforcement Version

|  |  |  |
| --- | --- | --- |
| **Runs** | **Honest wins** | **Malicious wins** |
| 20 | 13 | 7 |

Table 4: Cancel a Particular – Bitcoin Version

# Analysis

The results are way worse that expected, the Bitcoin protocol performed again better than the Reinforcement protocol. The results obtained for the Reinforcement Version are not good at all.  
It was easier to find a competing chain heavier than to find one longer.

Besides the selfish mining reason presented before there are other reasons that motivate these results.

Firstly, the malicious miners have more time to find a hash than the honest miners. When a propose block is committed by an honest node all the other honest nodes send their reinforcement and stop mining, waiting for the commit block. This makes it impossible to find a reinforcement with weight greater than 1. Also, this gives to the malicious nodes some extra seconds to find a nonce large enough if they didn’t find it yet. This time between the propose and the commit blocks was 1,5 seconds which is rather high when compare with the number of seconds to find a hash with seven nodes, roughly 15 seconds.

Secondly, the weight of a chain supported by x nodes is linear on expectation but it is affected by noise. As the honest miners follow always the heaviest chain, it is only necessary to have the malicious chain heavier than the honest one at one point to motivate a change. The difference between malicious and honest nodes in the experiment, 4 against 3 is very tiny and can be heavily affected by the noise. If the experience is run with more nodes, even in the same proportion 4:3, the relative noise is expected to decrease and better results should be obtained. A non-linear weight function with more emphasis in the number of reinforcements than in its quality can further help reducing the noise and enhance the power of the majority.

Thirdly, the OS might as well have had some influence in the results because the processes had to share the processing power of the same machine.

# Conclusion

Our implementation of the Reinforcement Protocol was outperformed by the Bitcoin protocol. The results are very far from the expected but, nonetheless, many questions about the protocol that allow a better understanding of it were raised and there are several points that can be improved:

* Selfish mining strategies can be implemented;
* Non-linear weight functions that give more emphasis to the reinforcement messages can be tested;
* The number of nodes should be increased to reduce the relative noise;
* The reinforcement threshold can be increased to increase the weight of the reinforcements and benefit the majority;
* Run the protocol distributed to avoid scheduling issue;
* Do a theoretical analysis to quantify the probabilities of the events tested.

Despite these results, we evaluate our work positively. We did what was proposed in the beginning of the project and produced a work that can be further continued and lead to even more interesting conclusions.

Notation: Along this document we will refer to the longest chain in the Bitcoin protocol as the heaviest. This is in fact true if we consider the weight of each block to be 1.

Malicious block – block proposed or committed by a malicious node

Honest block – block proposed or committed by an honest node (take out the quotation marks if the notation part is added)

Appendix

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| # 59 Fs and a 8 | |  |  |  |  |  |
| COMMIT\_TH = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF8 | | | | | | |
|  |  |  |  |  |  |  |
| # 60 and 8 Fs (16 times larger than commit TH) | | | |  |  |  |
| REINF\_TH = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF8 | | | | | | |
|  |  |  |  |  |  |  |
| SWITCH\_TH = 0 | |  |  |  |  |  |
|  |  |  |  |  |  |  |
| CANCEL\_PARTICULAR\_BLOCK\_TH = 60 | | |  |  |  |  |
|  |  |  |  |  |  |  |
| PORT = 5600 | |  |  |  |  |  |
|  |  |  |  |  |  |  |
| CANCEL\_BLOCK\_MIN\_RANGE = 2 | | |  |  |  |  |
|  |  |  |  |  |  |  |
| CANCEL\_BLOCK\_MAX\_RANGE = 4 | | |  |  |  |  |
|  |  |  |  |  |  |  |
| TRANSACTION\_INTERVAL = 3 | | |  |  |  |  |
|  |  |  |  |  |  |  |
| REINF\_TIMEOUT = 1.5 | |  |  |  |  |  |
|  |  |  |  |  |  |  |
| COMMIT\_TIMEOUT = 10 | |  |  |  |  |  |
|  |  |  |  |  |  |  |
| DELIVERY\_DELAY = 0 | |  |  |  |  |  |
|  |  |  |  |  |  |  |
| CLEAN\_PREVIOUS\_BLOCKS = 100 | | |  |  |  |  |