# 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 is 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 contain 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 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 and the search for nonces continues until hash module is explicitly stopped.