Collaborative Block Reinforcement in Blockchain Systems

Spring Semester – 16/17

Artem Shevchenko
André Águas

Advisors: Prof. Rachid Guerraoui
Matej Pavlovic

Table of Contents

Introduction

The article [1] raises the question of some issues of the standard Bitcoin protocol [2]: 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 in the article [1]. The main idea of this protocol is to use the proofs 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 which is supported by the largest cumulative amount of the proof of work is considered as valid (the heaviest one). This semester project is devoted to the practical implementation of the aforementioned protocol by using Python and asynchronous framework Twisted and to make experiments which will test its efficiency (in terms of solving the aforementioned issues) in comparison to the standard Bitcoin protocol.

Protocol Description

The short description of some concepts of the protocol, which are used in this project, is provided below. As was mentioned in the article [1], on the high level, the algorithm of the collaborative block reinforcement protocol is the following:

1. Miners do their work on top of what they believe is the last block of the blockchain.
2. During the mining, a 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 the 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 a new mining on top of a new block (Note that in the reinforcement protocol it is only possible to mine on top of committed blocks).

The scheme of the algorithm is shown on the Figure 1.

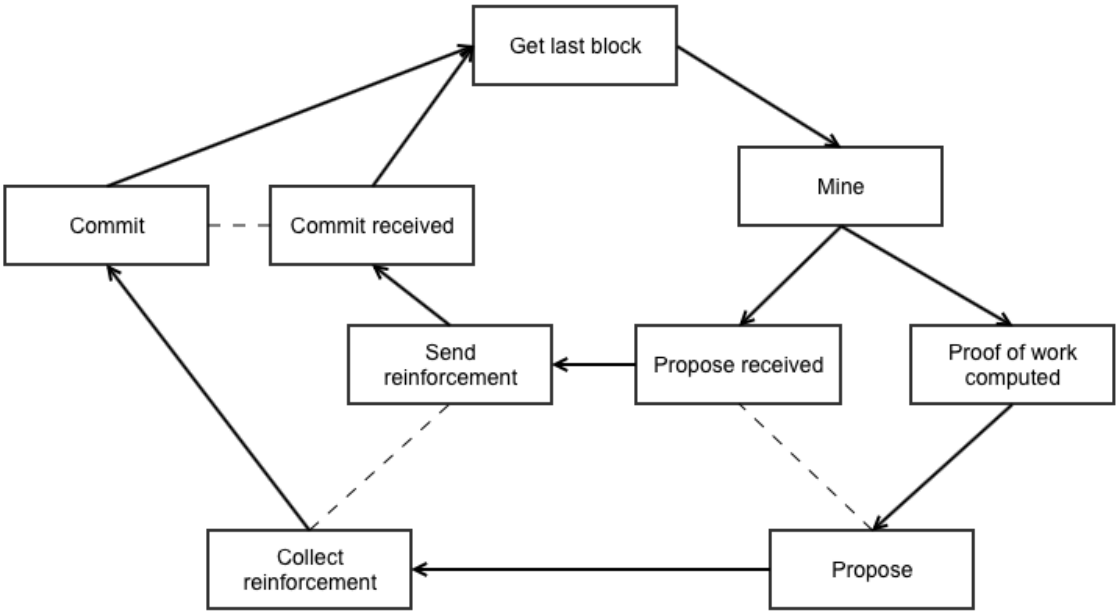

Figure 1: Scheme of the Algorithm

Some of the 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. 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, ***pub_key*** is miner's public key, ***A*** is the last block and ***d*** is the mining difficulty parameter for appending a new block. If miner does not

find a nonce that would produce a hash < $\boldsymbol{d}$, it may save nonces that produce considerably small hashes. These nonces, although are 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 of being rewarded, may reinforce both blocks (make double reinforcement). Such behavior must be prevented and for this purpose the concept of the proof of misbehavior (POM) is used. Any pair of messages where a 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 the malicious miner may claim the reward of the malicious miner by adding the POM to the block.

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

$$\boldsymbol{w = \frac{d}{h}}$$

where $\boldsymbol{h}$ is the hash corresponding to the nonce, $\boldsymbol{d}$ is the mining difficulty parameter for appending. The weight of the block is the sum of the weights of all POWs supporting this block (by definition, the weight of block cannot be less than 1). The weight of the chain is 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, the mining difficulty parameter for reinforcement ($\boldsymbol{d_r}$) 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: corresponding hash < $\boldsymbol{d_r}$).

Implementation Details

Event-driven Programming and Twisted

It was decided to use the event-driven 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 implementation of a distributed system in which nodes must react to asynchronous events such as network messages and finding 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. The Figure 2 illustrates the main loop of the reactor and triggering the callback function in case of event.

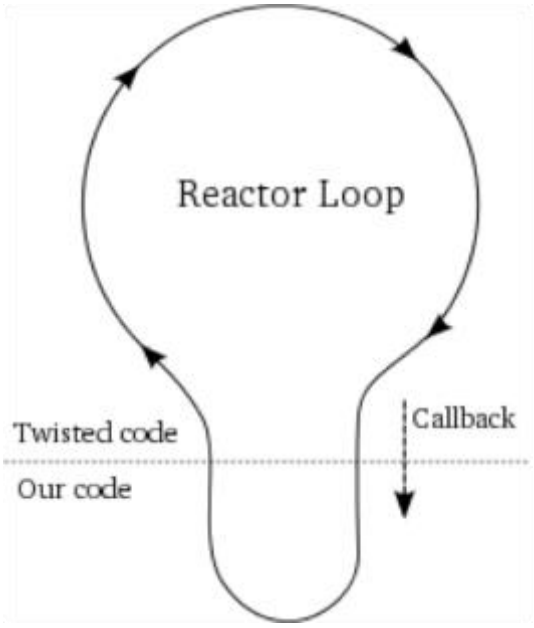

Figure 2: Reactor Pattern

Modules

The implementation of the protocol consists of modules each of which has a particular purpose (blockchain, broadcast, hash, states, the proof of misbehavior (POM), setup). The client that produces transactions is implemented as a separate program. The parameters of the implementation are: the timeouts of reinforcement collection and commit receiving, the threshold for the switching of mining, the mining difficulty for appending a new block and reinforcement, the number of blocks during which the POM is searched (***k*** in the protocol description), and some others which are necessary for the experiments (Appendix). The relations between the modules are shown on the Figure 3. This figure also illustrates the exchange of messages between nodes and client.

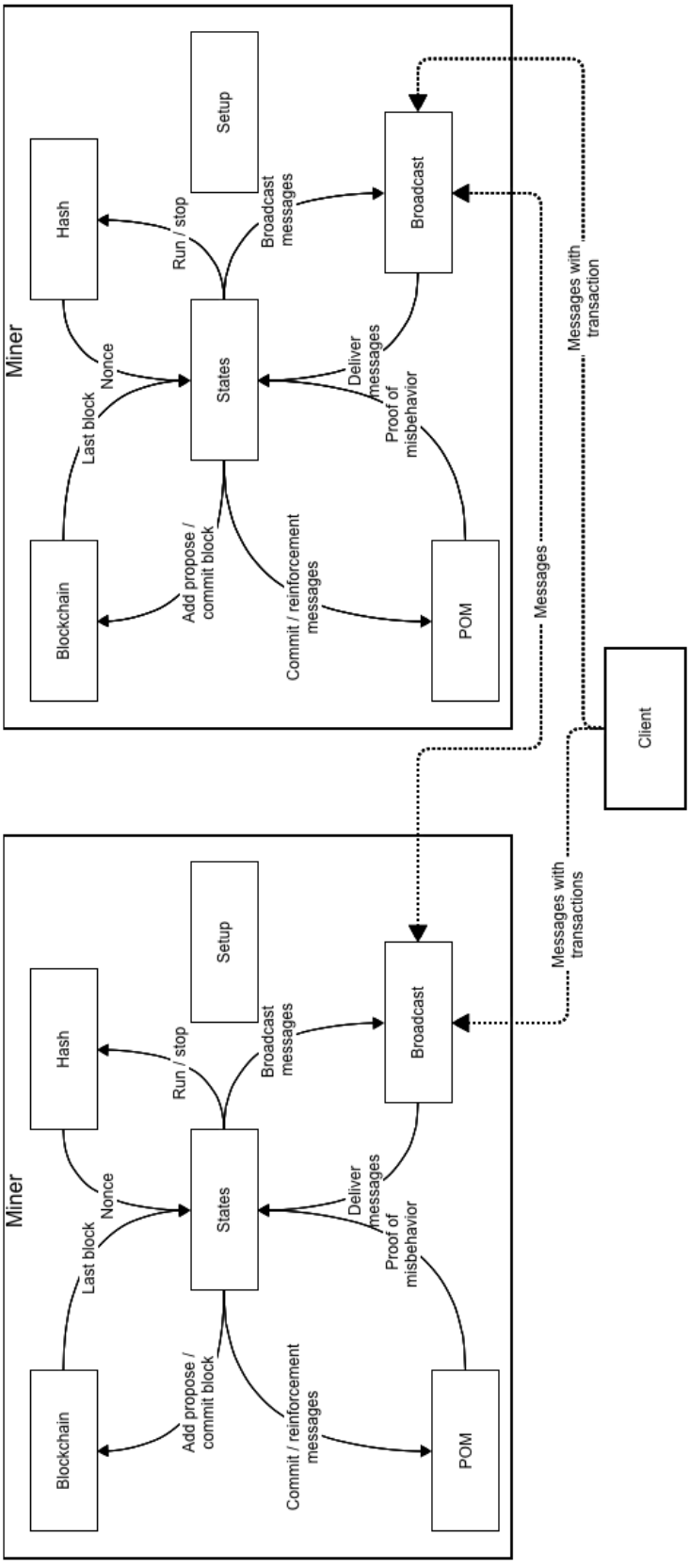

Figure 3: Structure of the Implementation

Blockchain

In the implementation of the blockchain two types of blocks are used: propose blocks and commit blocks. A propose block contains 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 is used during the experiments and shows if the block was added by a malicious miner. 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 that 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 a JSON formatted string and the ability to create blocks from a JSON formatted string is added.

The implementation of the blockchain allows the following main operations: addition of a block of any 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 the pair of blocks. For instance, first (genesis) propose and commit blocks have depth equal to 0, next propose and commit blocks have 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 is checked (i.e. the nonces produce small enough hashes). In order to make the experiments, there is the ability to return the block which is the last one in the heaviest malicious chain (i.e. a chain that terminates with a block with the malicious flag set). In order to deal with blocks which arrive out of order because of network issues, the pool of blocks is used. If the previous block for the just arrived block is not in the blockchain, the arrived block is added to the pool of blocks and is retrieved from it when the previous block is added to the blockchain. The structure of the blockchain is illustrated on the Figure 4.

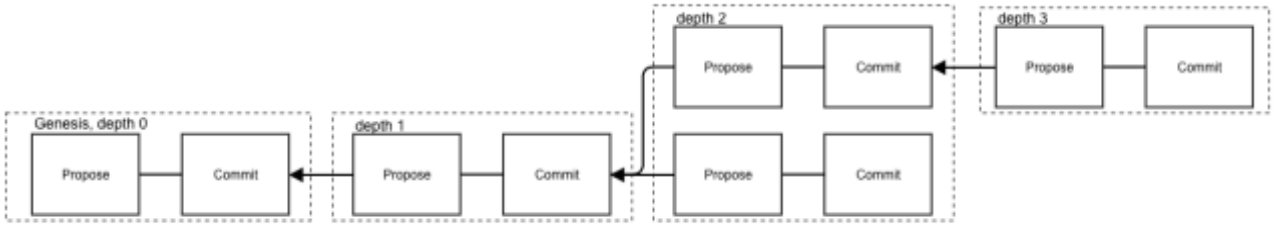

Figure 4: Structure of the Blockchain

States

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

When the mining starts, the finite-state machine is in the ***mining*** state. In this state, the following events are processed:

1. Nonces, found by hash module (see the description below). If the hash is sufficient for appending a new block to the blockchain (less than the mining difficulty parameter for appending mentioned above), the hashing module is stopped, a new propose block is created, it is appended to the 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 for reinforcement and the mining continues.
2. Propose messages, delivered by broadcast module (see the description below). 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 the current mining is going, the current mining is stopped by stopping the hash module and all the 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 below). 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 the current mining is going is greater than or equal to the threshold for the switching of mining (mentioned above as the implementation parameter), the current mining is stopped by stopping the hash module and a 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 corresponds to the propose which initiated the switching from the mining state to the reinforcement sent state, the state is switched to the ***mining*** state and a 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.

To avoid getting stuck in the reinforcement sent state due to network conditions or 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 a new mining on top of the last block of the blockchain starts.

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 the blockchain. Messages are also transferred to the POM module.
3. Reinforcement messages, delivered by broadcast module. The reinforcement messages are transferred to the POM module. The following processing is done only if the reinforcement message corresponds to the propose after which there was the switching from the mining state to the reinforcement collecting state. All nonces in the reinforcement message are checked if they produce small enough hashes. Those nonces which are valid by this criterion 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, a commit block including all collected reinforcements and the proof of misbehavior (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 are also included as reinforcements. The state is switched to the ***mining*** state and a new mining on top of the last block of the blockchain starts.

The diagram of states is shown on the Figure 5.

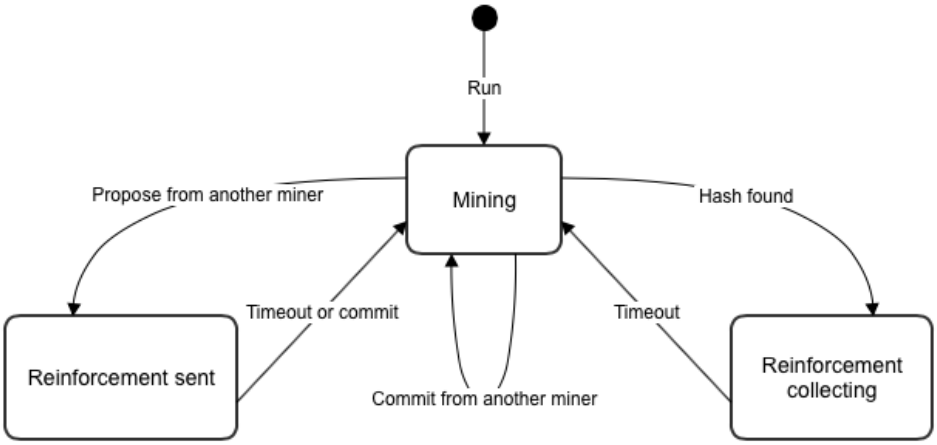

Figure 5: Diagram of States

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 new block to the blockchain (less than the mining difficulty parameter for appending mentioned above), hashing module is stopped, new propose and commit blocks are created, appended to the local copy of blockchain and broadcasted to all miners. A 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 the 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 the current mining is going is greater than the threshold for the switching of mining, the current mining is stopped by stopping the hash module and a 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 the mining difficulty parameter for reinforcement mentioned above. The hash is obtained by SHA-256 hashing miner's public key, nonce and the hash of the 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 the hash module is explicitly stopped.

Setup

This module generates the configuration files necessary to the correct behavior of the miners. These files are: one file for each miner and client containing its private key, and one shared file containing the configuration of the program (i.e. the timestamp of the genesis block, which miners are malicious and whether the Bitcoin or the Reinforcement protocol – we will refer by this name to the collaborative block reinforcement protocol further – should be run) and the

miners' and clients' public keys, IP addresses and ports. 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 state module, 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 messages passing models that this library supports, we used the "Publish – Subscribe" 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.

Every time a reinforcement or a commit message is delivered all the contained reinforcements 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 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 reinforcement messages in the next commit block that this node will broadcast.

To reduce the memory used by this structure, we use the fact that the blocks that are far from the last block of the blockchain will unlikely be forked. So we set a parameter ***k***, which defines the maximum distance from the last block of the blockchain on which we should store reinforcements. All the reinforcements that are deeper than ***k*** are deleted.

By default, no miner will double reinforce. So, to test our implementation we created a command line parameter "-f" (which means "faulty") 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 program continuously broadcasts 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 protocol with the Bitcoin protocol. We focused our experiments on the likelihood of forking the longest chain. A chain harder to fork will lead to a reduction of the number of blocks necessary to consider a block valid and will, consequently, reduce the latency of transaction confirmation.

The tests considered represent only some of the possible attacks to the protocol. In all the tests, there is a set of malicious and a set of honest miners. The honest ones always follow the protocol. The malicious ones have the goal of forking the main chain, so they often deviate from the protocol and follow the strategies described below.

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 blocks on top of which it is mining. So, further analysis of these logs can be done afterwards.

Test Conditions

To have runs as independent as possible we generated new keys and a new genesis 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 OS X El Capitan 10.11.6 with a 2.2GHz quad-core Intel Core i7 processor.

There was a total of 7 nodes, out of which 4 were honest mines and 3 were malicious.

The exact configuration constants used in the program is included in the Appendix.

Test Cases

Cancel Every Block

In this configuration, the malicious miners try to cancel every 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 it to 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.

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, as illustrated in the Figure 6. If the malicious chain is the heaviest, the malicious miners have successfully cancelled the block. At this point all the nodes will restart mining on top of the heaviest chain and the malicious miners will try to cancel the next block, the block at depth 2.

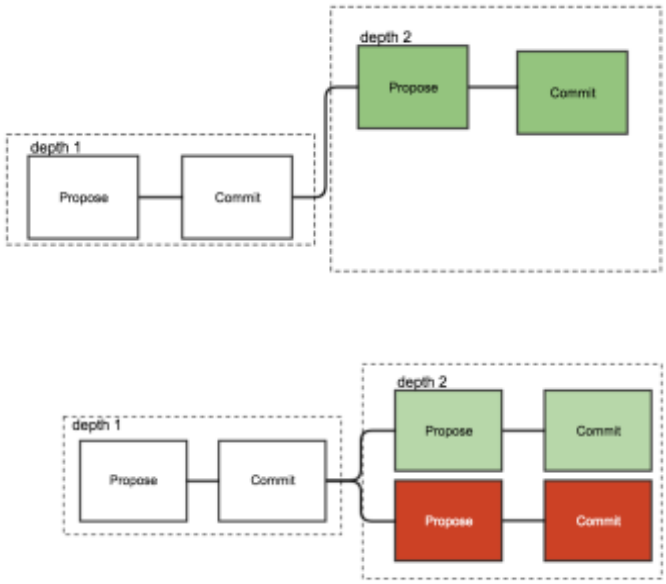

Figure 6: Outcomes of the Cancel a Particular Block Strategy

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. In the best case for the malicious miners, both the malicious and the honest chains will have weight 1. To break the tie, the winning miner is the one that finds the second block first. The possible outcomes of one iteration are included on Figure 7.

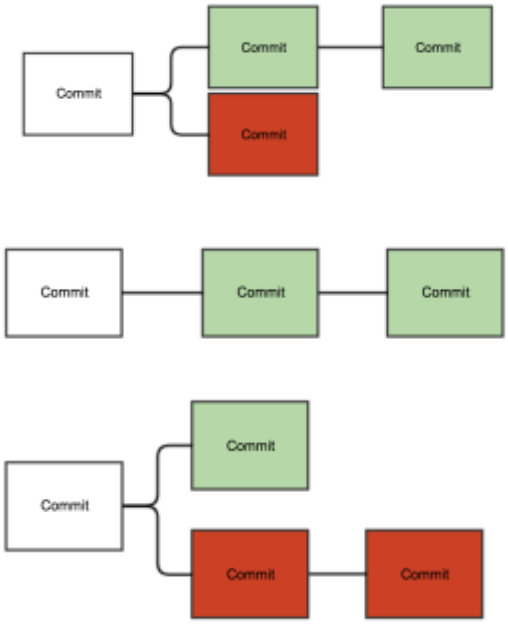

Figure 7: Outcomes of the Cancel Every Block Strategy – Reinforcement Version

In the first two diagrams on the Figure 7 the winning chain is the honest one, whilst in the last diagram 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 version 5 times. In each run the malicious miners had the opportunity to cancel 25 blocks, which means that the Reinforcement version runs until the depth 25 and the Bitcoin version until the depth 50. The results of the experiment for the Reinforcement protocol are represented on Table 1 and the results of the experiment for the Bitcoin protocol on Table 2.

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.

One of the reasons that motivates this difference is 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 increase 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 2) 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 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 shared 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 it to the others and wait for the honest node to commit a block. Then, the malicious nodes propose a block, reinforce it 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 when 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 not recoverable with very high probability).

We expect the malicious miner to be more successful in this test than in the cancel every block test because they do not give up if they does not cancel the block immediately, they carry on.

Results

We ran each version 20 times. The depth of the block to cancel was randomized between runs. The results of the experiment for the Reinforcement protocol are represented on Table 3 and the results of the experiment for the Bitcoin protocol on Table 4.

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 than expected, the Bitcoin protocol performed again better than the Reinforcement protocol. The results show that is very easy to fork the blockchain in the latter protocol.

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

Firstly, the malicious miners have slightly 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 reinforcements and stop mining, waiting for the commit block. This makes it impossible to find a reinforcement with weight greater than 1. Also, this gives the malicious nodes some extra time 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 if we compare it 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 represented on Figure 8. 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, which is also illustrated on Figure 8. 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 experiment 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 on the quantity of reinforcements than on their quality can further help reducing the noise and enhance the power of the majority.

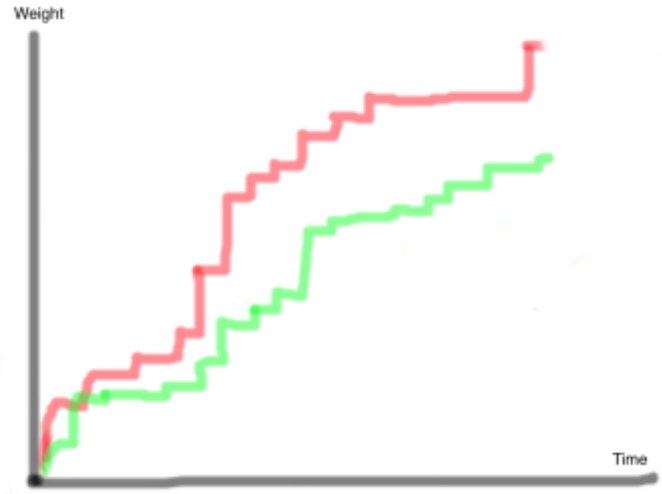

Figure 8: Influence of Noise on the Weight of Chains (Green: Chain supported by 3 nodes; Red: Chain supported by 4 nodes)

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 ones 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 can be tested;
- The number of nodes should be increased to reduce the relative noise;
- Run the protocol distributed to avoid scheduling issues;
- Do a theoretical analysis to quantify the probabilities of the events tested and the impact of the reinforcement threshold.

Despite the obtained results, we evaluate our work positively. We did what was proposed at the beginning of the project, came to some conclusions and made a work that can be further continued. We believe that the Reinforcement protocol can be improved and, in a next version, obtain similar or better results in comparison to the Bitcoin protocol.

References

[1] PAVLOVIC, M. Collaborative Block Reinforcement in Blockchain Systems

[2] NAKAMOTO, S. Bitcoin: A peer-to-peer electronic cash system. http://www.bitcoin.org/bitcoin.pdf, 2008.

Appendix

The content of src/Contants.py:

```
COMMIT_TH =
0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF8

# 16 times larger than commit TH
REINF_TH =
0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF8

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
```