TinyCoin: Simulating mining strategies in a simplified Bitcoin Network

Final Project for the Peer to Peer course

Francesco Balzano
Master Degree in Computer Science and Networking
A.Y. 2016-2017

Contents

1 Overview

In this project I have implemented a simplified version of Bitcoin, called TinyCoin, whose specifications are reported in [1]. In brief, TinyCoin distinguishing features are:

- each user has a single address that records the unspent amount at that node
- each transaction has a single input, a single output and does not include neither a digital signature nor scripts
- network nodes may be either normal nodes or miners. In turn, miners may be either honest or fraudolent (*i.e. Selfish Miners*). Each miner has a type that reflects its mining hardware: CPU, GPU, FPGA or ASIC
- there is a centralized oracle that decides which miner has created a new block of the blockchain at regular intervals of time. The decision is biased by the computational power of the miners. Each block is unique.

The goal of this project is to evaluate the selfish mining strategy defined in [2] in TinyCoin. This strategy is evaluated by taking into account different metrics and parameters. The results and the discussion of this experiment are reported in the *Results* section.

2 Design choices

In the following subsections I explain the design choices that I made. This is a high level description, indeed the classes of the project are described in the *Implementation* section.

2.1 Network Nodes

Any node in the TinyCoin network is either a (normal) node, or a (honest) miner or a selfish miner. According to the specifications, I have considered a static topology, which is generated at random at the beginning of the simulation.

- **node**: makes and receives transactions
- **miner**: makes and receives transactions and mines blocks of the blockchain. In particular, as soon as a new block is mined it is immediately advertized to all the nodes in the network.
- **selfish miner**: makes and receives transactions and mines blocks of the *private* blockchain. A selfish miner indeed holds both a copy of the public blockchain, which is the "official" blockchain, and a copy of the private blockchain, which is the blockchain created and maintained collectively by all the selfish miners. At any time the two blockchains may be equal or may differ for some block. If they differ, it is because the selfish miners have discovered new blocks which have been added to the private blockchain but have not been disclosed to the public. Indeed when a selfish miner mines a new block, it does not naively publish it and add to the public blockchain. Instead, it applies a strategy that allows it and the other selfish miners to get the maximum revenue from their computing power. One of the strategies that they can follow is explained in [2]. In this project, I chosed to implement it. The pseudocode is reported in algorithms 1, 2 and 3.
  Initialization Since we assume that all the nodes are present at the beginning of the simulation, at initialization (algorithm 1) a selfish miner has both the private and public blockchain empty, and the *privateBranchLength*, which tells how many blocks the two blockchains differ for, is set to zero.
  Selfish Miner found a block Algorithm 2 describes what happens when one selfish miner creates a new block: the block is added to the private blockchain and the *privateBranchLength* is incremented accordingly. If there was a fork in the public blockchain (line 4), then the selfish miners are in competition with the honest miners to insert their own blocks in the final version of the blockchain. In this case the new block is advertized to all the nodes of the network. In so doing, the selfish miners are quite confident that their two blocks will be included in the blockchain, at the expense of the block mined by the honest miner. Figures 1, 2 and 3 illustrate this situation.
  Other miners found a block Algorithm 3 describes the behaviour of the selfish miner when a (honest) miner creates a new block. If the private blockchain and the public one are equal (line 3), then the selfish

miner is fallen behind the other miners and thus accepts the new block and continues mining on top of it. If the selfish miner was one block ahead (line 6), then it advertises to all the nodes its private block, leading to a fork in the blockchain. If the selfish miner was two blocks ahead (line 8), then it publishes both its blocks. This way, it is quite confident that these two blocks will become part of the blockchain, because the branch that it creates is longer than the branch created by the honest miner. If the selfish miner had an advantage of more than 2 blocks (line 11), then it publishes only the first unpublished private block, leading the honest miners to spend clock cycles to mine blocks that at the end won't be part of the final blockchain.

2.2 Blockchain, Blocks and Transactions

- **transaction**: transactions are generated by nodes. They have an input node, an output node, an amount and a fee. For the sake of simplicity, I have assumed that each node pays the transaction at the moment of creation, and that for each transaction the fee is fixed in percentage.
- **block**: a block basically contains a list of transactions. For the sake of simplicity, I assumed that the reward for the block creation is immediately given to the miner. If such block is later removed from the blockchain, then also the reward given to the miner is given back. The block creation is arbitrary: it is not required to the miners to solve any Proof Of Work, instead it is the oracle that chooses at random the miner that has to create the new block of the blockchain.
- **blockchain**: a blockchain is a list of blocks. It does not include hash pointers, so it is not tamper-free. A block is added to the blockchain if it has been created by a miner and if its parent matches the current last block of the blockchain.

2.3 Forks

Forks have a serious impact on the selfish mining strategy. Indeed, selfish miners goal is basically to fork the blockchain, creating a branch that eventually becomes longer than the "official" one and thus gets included in the blockchain, to the detriment of the branch created by honest miners. There are two kinds of forks that are supported in this project:

- **selfish miner**: it supports forks of arbitrary length, that is the public and the private blockchain may differ for an arbitrary number of blocks from the point of view of the selfish miner. This fact is necessary in order to apply its strategy, as described in algorithms 1, 2 and 3.
- **node**: a normal node only supports a simplified kind of fork, that is one in which there are two branches made of only one block. The next block that is received by the node that extends one of the two branches actually resolves the fork. If at this point another blocks arrives willing to extend the abandoned branch, it will be simply discarded. There are two reasons for this assumption. The first is simplicity: at any time there are no orphan branches in the blockchain, but only a simple fork is admitted. The second is tamper resistance: since the blockchain is not tamper-free, avoiding to leave branches in the blockchain and allowing to fork only the last block of the blockchain means that fraudolent miners cannot mine on top of arbitrary branches instead of the "official" one.

2.4 Network Latencies

Algorithm 1 Selfish Miner initialization

1: public chain $\leftarrow \emptyset$
2: private chain $\leftarrow \emptyset$
3: privateBranchLength $\leftarrow 0$
4: Mine at the head of the private chain

Algorithm 2 Selfish miners found a block:

$\Delta_{prev} \leftarrow$ length(private chain) - length(public chain)
append new block to private chain
privateBranchLength $\leftarrow$ privateBranchLength + 1
if ($\Delta_{prev} = 0$ AND privateBranchLength $= 2$) **then**
    private chain $\leftarrow$ public chain
    privateBranchLength $\leftarrow 0$
Mine at the new head of the private chain

Algorithm 3 Honest miners found a block:

$\Delta_{prev} \leftarrow$ length(private chain) - length(public chain)
append new block to public chain
if ($\Delta_{prev} = 0$) **then**
    private chain $\leftarrow$ public chain
    privateBranchLength $\leftarrow 0$
else if ($\Delta_{prev} = 1$) **then**
    publish last block of the private chain
else if ($\Delta_{prev} = 2$) **then**
    publish all of the private chain
    privateBranchLength $\leftarrow 0$
else
    publish first unpublished block of the private blockchain
Mine at the new head of the private chain

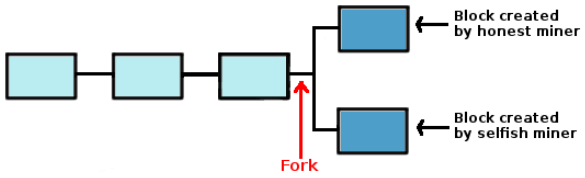

Figure 1: When two different blocks are generated having the same parent (usually these two blocks are generated very close in time), a fork is created in the blockchain.

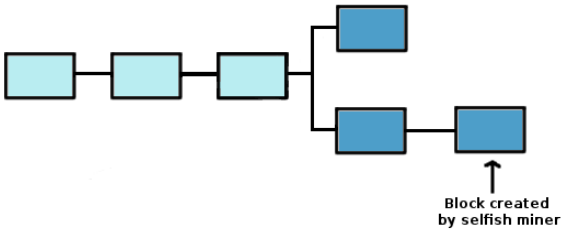

Figure 2: When a selfish miner creates a new block, it adds it immediately to the public blockchain if and only if there is a fork in the blockchain. This way, selfish miners are one block ahead with respect to other miners, and so the probability that their two blocks will be part of the final blockchain increases.

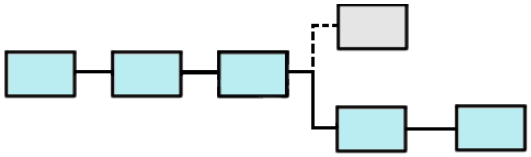

Figure 3: Selfish miners have reached their goal: their branch is the longest, and so all the miners that were mining on the shorter branch abandon it and start mining on the other, longer branch.

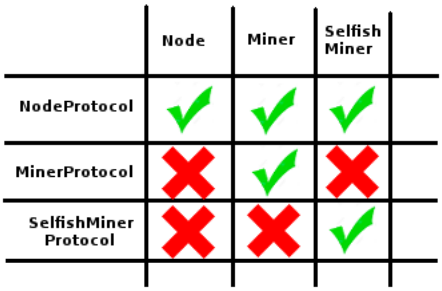

Figure 4: Relationships between node types and node protocols.

3 Implementation

I set up the simulation using the event driven version of the Peersim simulator. The classes of this project are in some way dictated by the framework itself. For instance, I wished to have three different classes Node, Miner and SelfishMiner that extended the base class NetworkNode. Each network node would have been one among Node, Miner and SelfishMiner, so that it was immediately clear which was its role in the network. Anyway this was not possible because I had to follow the structure of classes defined by the framework. In this section, I describe the classes that I developed with reference to their role in the framework.

3.1 Protocols

Every node in the network is a TinyCoinNode. If the node is a normal node, only the NodeProtocol is enabled; if it is a miner, both the NodeProtocol and the MinerProtocol are enabled; if it is a SelfishMiner, both the SelfishMinerProtocol and the NodeProtocol are enabled (figure 4).

- **NodeProtocol** The NodeProtocol is responsible for creating and advertising transactions and for publishing blocks of the public blockchain. The method *nextCycle()*, called at every cycle, creates a transaction with a fixed probability defined in the configuration file. Both the amount and the recipient of the transaction are random. Then the transaction is published on the network. The method *processEvent()* is invoked every time the NodeProtocol receives a message. A message may be either a transaction or a block. In the former case, if the node hasn't already seen it, it adds the transaction to its transaction pool and floods it to its neighbors. In the latter case the behaviour is different based on the type of node. If the node is a normal node or a honest miner, it checks whether the received block has to be added to the blockchain, checks the eventual presence of forks, makes some other operations and finally floods the block to all its neighbors. Else, if the node is a selfish miner it implements the algorithm 3. The fact that also the selfish miners run the NodeProtocol could seem weird, but actually there are two reasons for this choice. First: a selfish miner is also a node, able to make and receive transactions. Second: I use the NodeProtocol as a "public layer" to exchange public information among the nodes of the network, namely transactions and blocks of the public chain. Instead, I use the SelfishMinerProtocol as a "private layer" to exchange private information inside the selfish miners pool, namely the blocks of the private blockchain not disclosed to public yet.

- **MinerProtocol** The MinerProtocol is run by the miners. At each cycle it is invoked the method *nextCycle()*, which is actually executed only if the oracle selected this miner for the current cycle. If this is the case, the miner picks a set of transactions from its transaction pool and creates a new block. The block is immediately advertised to all its neighbors and the revenue for mining a new block is immediately assigned to the miner.

- **SelfishMinerProtocol** The SelfishMinerProtocol is run by the selfish miners. At each cycle it is invoked the method *nextCycle()*, which is actually executed only if the oracle selected this selfish miner for the current cycle. This method creates a new block using the transactions in the transaction pool and then implements the algorithm 2, that is either shares the block with the selfish miners or with all the nodes based on convenience. The method *processEvent()* is invoked every time a "secret" block is received from a selfish miner, so it simply implements the algorithm 2.

3.2 Initializer

- **NodesInitializer** The NodesInitializer initializes the nodes in the network based on the probability values received from configuration file. Namely, each network node has a given probability of being either a normal node, a miner or a selfish miner. If it is a miner or a selfish miner, it has a given probability of mining using either CPU, GPU, FPGA or ASIC.

3.3 Control

- **TinyObserver** The TinyObserver monitors the simulation, logging on disk statistics that will be later plotted. These statistics include the number of forks occurred in the simulation, the number of blocks in the blockchain that have been mined by honest and fraudolent miners, and so on.

- **Oracle** The method *execute()* of the Oracle is invoked once per cycle. This method selects at random the miner that has created the new block. In some (rare) cases, the oracle may decide that two miners have concurrently solved the Proof of Work, and so two different blocks will be mined such that they will result in a fork of the blockchain. This may happen in practice, so I decided to model this case. At each cycle, each miner has a probability of being selected that depends on its computing power: the more the power, the more the probability that such miner has solved the Proof of Work. During the first invocation, the oracle initializes the probabilities of selecting a miner of a given type, taking into account the hash rate of each miner type and the total hash rate of the network. The hash rate of each miner type is a parameter that is defined in the configuration file. The total hash rate of the network depends on the number of miners in the network and on their type. The random selection of a miner is done in two steps. In the first, it is randomly chosen the type of the miner. In the second, the nodes in the network are shuffled and then they are scanned in this new order, taking the first miner with appropriate type.

3.4 Other Classes

In this paragraph I describe the classes of the project that don't belong to the previous categories.

- **Transaction** A Transaction is a simple class having one input node, one recipient node, a reward to be paid to the recipient and a fee to be paid to the miner that includes the transaction in the block.

- **Block** A Block contains a set of transaction. It is identified by a unique identifier and specifies the parent block of the blockchain, the miner that created the block and the reward that such miner has earned for creating the block.

- **TinyCoinNode** Every node in the network is a TinyCoinNode. A TinyCoinNode has a type (NodeType) and, in case it is a miner, a MinerType. Then it maintains a copy of the public blockchain, a transaction pool holding the unconfirmed transactions (the ones not yet inserted in a blockchain block) and the current balance of the node.

- **NodeType** It is a enum that indicates the type of each node. The type can be either Node, Miner or SelfishMiner.

- **MinerType** It is a enum that indicates the type of a miner. It can be either CPU, GPU, FPGA or ASIC.

4 Results

alala

5 Conclusion

6 Limitations

- **forks**

7 References

1. TinyCoin: Simulating mining strategies in a simplified Bitcoin Network
2. Majority is not Enough: Bitcoin Mining is Vulnerable