P2P Final Project 2016/2017: TinyCoin: Simulating fraudulent mining strategies in a simplified Bitcoin Network

Federico Errica

1 Introduction

This report is intended to show the main design choices adopted in the implementation of TinyCoin as well as the impact that fraudulent mining strategies have on the overall structure of the blockchain. Selfish mining [1] is the only strategy analysed, and we study what happens either when dishonest miners join together in a big pool or when they act alone. Interestingly, results are in line with theoretical ones, but we also show what we get when it comes to adding latency and varying configuration parameters.

2 Implementation notes

In this section we describe the main design choices, which can be observed in the class and activity diagrams. First of all, the entities that play an important role in the project are shown in Figure 1. A TinyCoinInitialiser is called by Peersim before the start of the simulation and decides if a

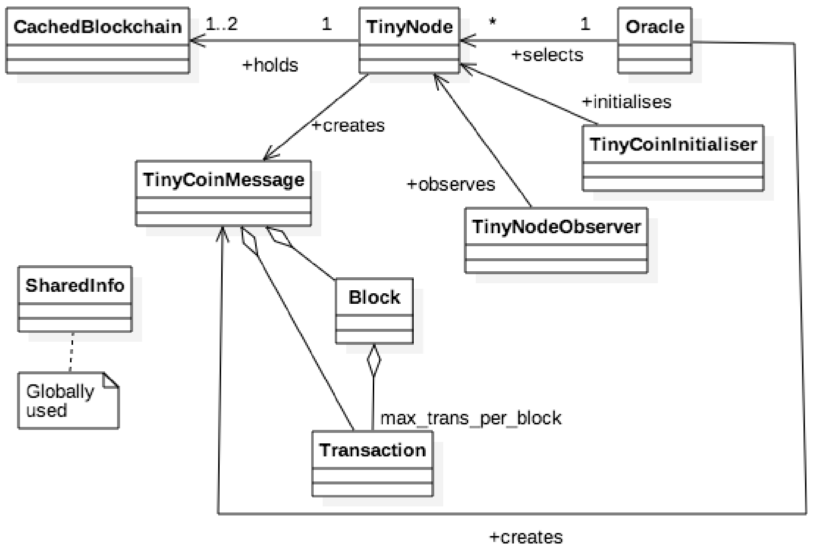

Figure 1: Class diagram

node is normal or miner; in the latter case it chooses the kind of mining power and if the selfish strategy must be adopted. The TinyNodeObserver is run almost at the end of the simulation and computes the statistics presented in Section 4. A Block is made of transactions, its unique id and the predecessor's id. A Transaction reports the source, the destination and the amount of bitcoins involved in the exchanges. This assumption has an important impact on the next implementation details. The Oracle decides when a miner should mine a block, and the decision is sent without any kind of delay to the recipient. A TinyCoinMessage encapsulates the kind of message, either MINED, BLOCK or TRANSACTION. SharedInfo manages the values stored by the user in the Peersim configuration file. Finally, the most important classes are TinyNode, which implements the node/miner/selfish behaviours, and CachedBlockchain which stores a copy of the local blockchain (which in this case corresponds to blocks with output transactions only) and something that in this project is called UTXO but in reality is a mapping from a node to its current bitcoins (since output transactions exactly describe how much bitcoins a node can spend). This is the most crucial choice in this project: each node has a local view of how much bitcoins any node can spend, and it uses this information to accept or refuse a transaction. To that aim, a CachedBlockchain stores a UTXO associated to the local blockchain, and a temporary one which is modified by incoming transactions. This class also handles blocks whose predecessors have not been received yet, and any number of parallel forks can be handled. A selfish miner stores 2 local blockchains, a public and a private one.

2.1 Oracle's activity diagram

The diagram depicted in Figure 2 points out the steps taken by the oracle. The only significant thing here is the fact that a miner is chosen, in order to build a block, according to a uniform distribution over the different available hardware.

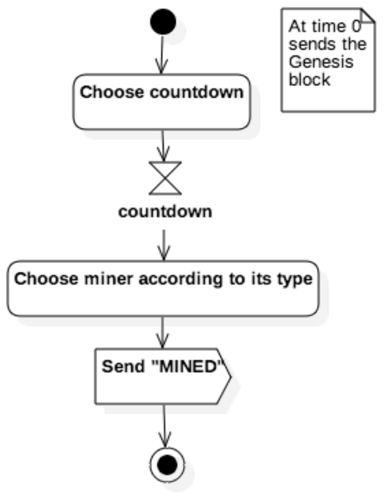
At time 0 sends the Genesis block
Choose countdown
countdown
Choose miner according to its type
Send "MINED"

Figure 2: Oracle's activity diagram

2.2 Normal node's activity diagram

A node's activity can be divided in two: one behaviour is controlled through the cycle-driven interface of Peersim, and it is responsible for generating a transaction and broadcasting it to the neighbours, whereas the other is triggered in response to a message, which can be either a transaction or a block (the MINED message is not handled). To accept a transaction, it is mandatory that the node has not received it yet and the transfer that is requested can be done. To implement this behaviour, a unique transaction IDs have been introduced: if conditions are met, the node updates the UTXO (by subtracting from the sender and adding to the recipient) and broadcasts the transaction. A memory pool of transactions is kept to figure out what transactions

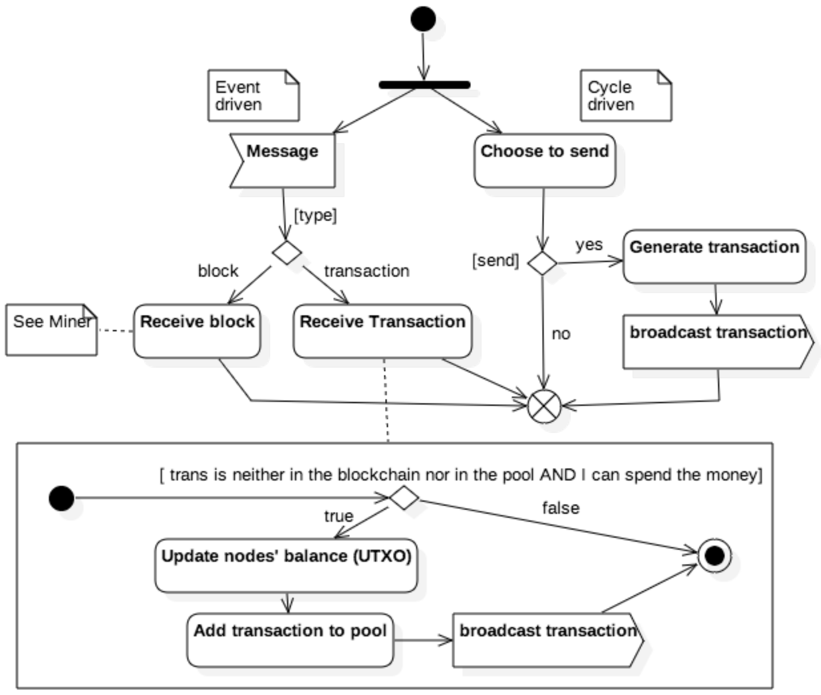

Figure 3: Normal node's activity diagram

have been accepted but not included in a block. In case a transaction which is included in a received block was not known before, the program checks that it is valid w.r.t the current blockchain. This implies that every accepted transaction in the local memory pool will need a second check before being inserted into a new block (and consequently into the blockchain), because what we called the temporary UTXO could not take into account an unknown transaction, and this would have resulted in inconsistencies. Apart from that, we also keep the transactions currently into the blockchain, no matter if they belong to an orphan block.

2.3 Miner's activity diagram

The mining activity is fundamental, and it includes a number of particular cases that need to be handled. When a miner is chosen by the Oracle, it has to build a block with the accepted transactions in the memory pool. It does so by picking them in the order of arrival. This should be enough in many cases, but as we previously said we may have not received a particular transaction which now belongs to the blockchain, so we carefully pick only those transactions which are valid in that moment. Looking at Figure 4 it is clear that the most important part for a miner is receiving a block. It is required that an already received block should be discarded and not propagated anymore, as in the case of transactions. Furthermore, if the predecessor has not yet arrived, we keep the block until we can correctly append one after another. We do so by recursively calling the

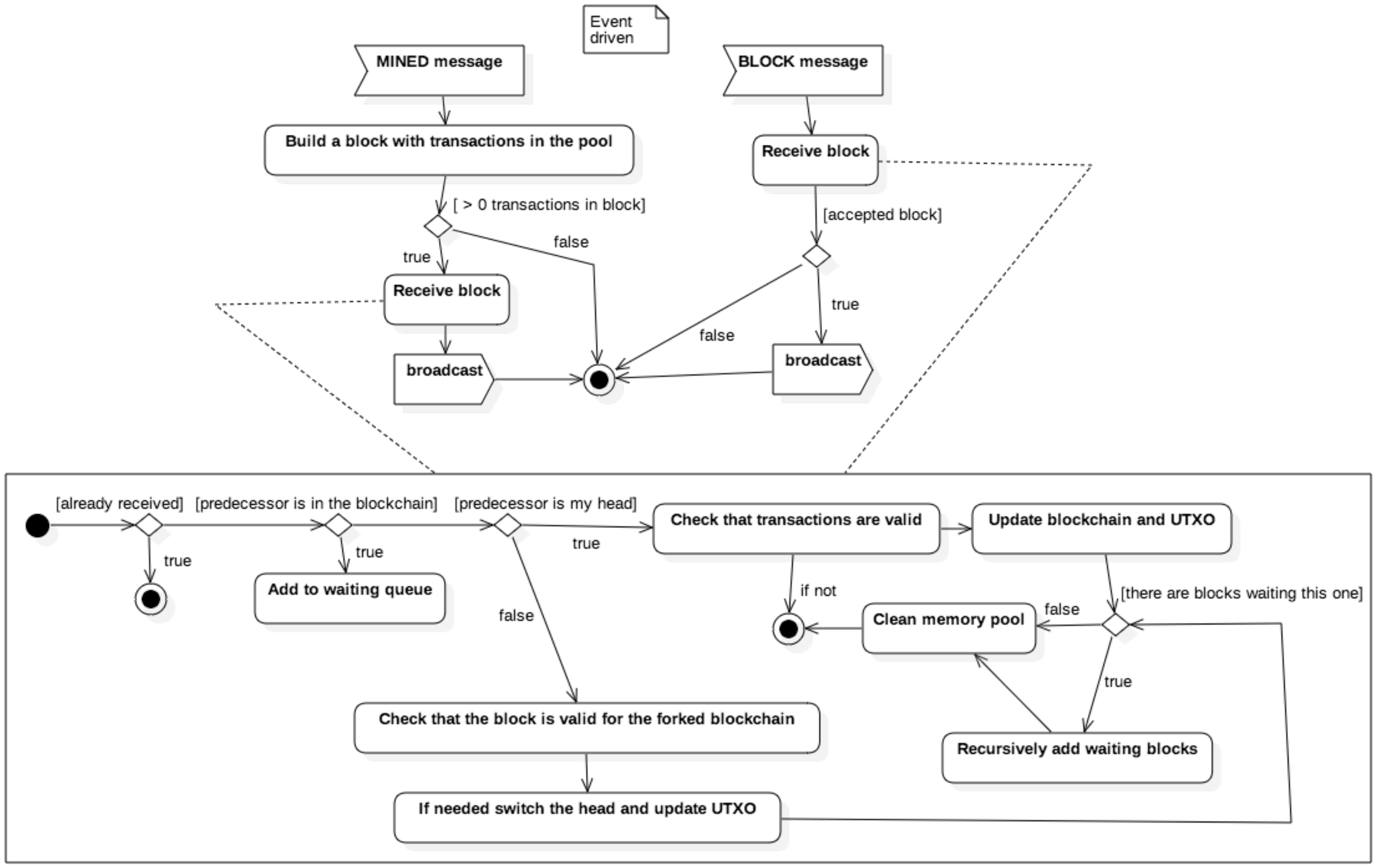
Event driven
MINED message
BLOCK message
Build a block with transactions in the pool
Receive block
[> 0 transactions in block]
false
true
Receive block
broadcast
[accepted block]
true
false
broadcast
[already received]
[predecessor is in the blockchain]
[predecessor is my head]
Check that transactions are valid
Update blockchain and UTXO
true
true
true
Add to waiting queue
false
if not
Clean memory pool
false
[there are blocks waiting this one]
true
Check that the block is valid for the forked blockchain
Recursively add waiting blocks
If needed switch the head and update UTXO

Figure 4: Miner's activity diagram

receiveBlock function, so as to handle also chains of waiting blocks. Then, if we need to concatenate the new block to the current head of the local blockchain, the algorithm validates the block, and updates the UTXO; finally, transactions' IDs are moved from the memory pool to another data structure. In case we are not extending the head, this means we are on a forked branch. To this purpose we need to efficiently compute the forked UTXO and accept a block based on that. If the forked branch surpasses the main one, a switch takes place.

To conclude, the selfish mining is almost exactly implemented as in [1]. The code should speak for itself.

3 Experiments

Peersim has been used to implement the TinyCoin protocol. Its event-driven version allows to simulate latencies when delivering a message, and the resulting graph topology is undirected and weakly connected with high probability. We describe the configurations tried in Table 1. Please notice that the degree gets doubled due to the fact that the graph has to be turned in an undirected one. We have chosen to vary the latency to send a block over the TinyCoin network, the percentage of selfish miners and the probability to choose one specific type of miner to mine a block (the computational power). Moreover, the introduction of a maximum number of selfish/asic miners allows for the creation of a simulated selfish pool made of a single miner with an arbitrary computational power. Finally, we noted that the oracle's gaussian distribution and the latency are in strict relation with each other when it comes to forks' creation, since they influence the number of blocks that concurrently travel the network. Thus we fixed the average time to mean a block to 600 cycles, which should correspond to 10 minutes in our view. Standard latency values reflect those in $https://blockchain.info/stats$

4 Results

First of all, we show what happens when block latencies are increased up to the order of a block creation time (with no selfish miners). We assume that a Peersim cycle corresponds to a second in the real world. In Figure 5 two interesting things happen: the former is the abrupt increase in the number of forks when we reach a block latency of 1/3 of the average mining time, and the topology of the network is made by 500 nodes with average degree 6. The latter is the threshold that we reach when we approach the block creation time, which is exactly the half of the mined blocks (500); this suggest the head of the blockchain is continuously forked. When doubling the latency again, results seem unpredictable. The second interesting result compares the percentage of individual selfish miners with the number of forks in the network. Indeed, in Figure 6 it can be observed that the more selfish miners in the network the more the forks, with some exception due to the variance of the results. This happens because many selfish miners tend to keep for themselves the blocks they find, and as soon as an honest miner publishes they tend to create forks so as not to lose their reward. The same reward has the effect to introduce new Bitcoins in the network, but it has no consequence on the selfish mining strategy.

In Table 1, we finally present the set of tests that have been run. To make an example regarding the variation of computational power, by assigning the same probability 0,3 to cpu, gpu etc. miners to be selfish we get a total selfish computational power of 30%. Each test is represented by a column, and we can immediately see how significant increases in the latency lead the blockchain (also in these cases) to have an incredibly high number of forks. However, the average length of each forks cannot be big, since selfish miners do not cooperate, and this is reflected in the number of dishonest

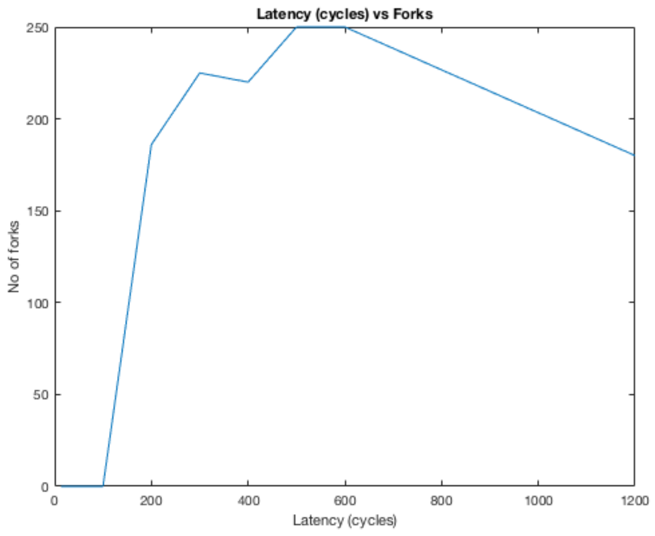

Figure 5: Latency vs forks (500 blocks to mine).

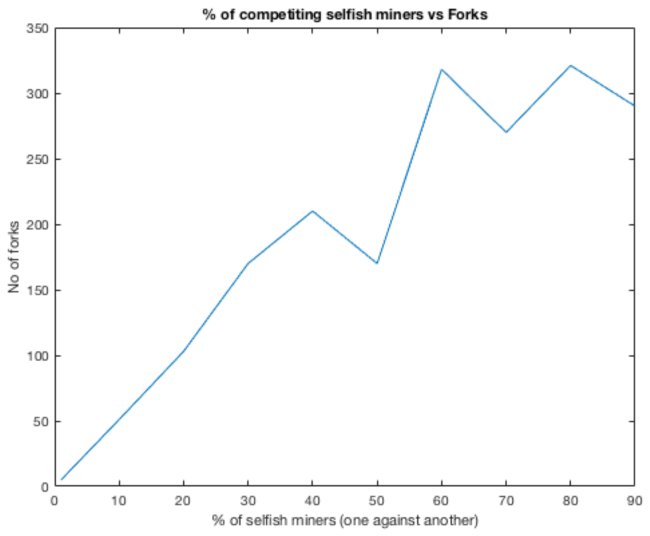

Figure 6: % of selfish miners vs forks (500 blocks to mine). The miners are not organised into pools.

miners currently rewarded. We observe that increasing the percentage of selfishness while keeping standard latencies achieves more or less the same effect, and this is in accord with the idea that selfish mining tend to create disorder when applied by small pools, so it is neither worth to be done. We then asked ourselves what happens when all miners are dishonest: since each of them would keep mined blocks for itself, we slightly modified the algorithm, just for this test, so that they publish when more than 10 blocks are mined. Then, as soon as a miner published the first block, everyone started doing the same. This resulted in a considerable number of forks, a very short blockchain (44 blocks against a maximum of 830) and a huge waste of resources.

Cycles	500000	500000	500000	500000	1000000	1000000	1000000	500000	100000
Size	500	500	500	500	500	500	500	500	500
Degree	3	3	3	3	3	3	3	3	3
Initial bitcoins	100	100	100	100	100	100	100	100	100
Send a transaction (%)	0.01%	0.01%	0.01%	0.01%	0.01%	0.01%	0.01%	0.01%	0.1%
Block reward	10	10	10	10	10	10	10	10	10
Extra reward per trans	5	5	5	5	5	5	5	5	5
Oracle mean	600	600	600	600	60	60	60	600	600
Oracle variance	50	3	50	50	50	50	50	50	50
No of transactions in a block	10	10	10	10	10	10	10	10	100
Block latency	3	50	3	3	0	0	3	3	3
Extra latency per trans	1	30	1	1	0	0	1	1	2
Normal node (%)	75%	75%	75%	75%	75%	75%	75%	75%	75%
Miners (%)	25%	25%	25%	25%	25%	25%	25%	25%	25%
Choose cpu (%)	5%	5%	5%	5%	5%	5%	5%	5%	5%
Choose gpu (%)	10%	10%	10%	10%	10%	10%	10%	10%	10%
Choose fpga (%)	30%	30%	30%	30%	30%	30%	30%	30%	30%
Choose asic (%)	55%	55%	55%	55%	30%	40%	51%	55%	55%
Max Asics	500	500	500	500	1	1	1	500	500
Selfish cpu (%)	30%	30%	60%	0%	0%	0%	0%	1%	0%
Selfish gpu (%)	30%	30%	60%	0%	0%	0%	0%	1%	0%
Selfish fpga (%)	30%	30%	60%	0%	0%	0%	0%	1%	0%
Selfish asic (%)	30%	30%	60%	0%	100%	100%	100%	1%	0%
Max selfish	500	500	500	500	1	1	1	500	500
Results									
Forks	107	394(+/-4)	473(+/-4)	0	299	321	187	238	66
Blockchain height	723	312	336	831	1300	1187	826	44	93
Avg forks' length	1.01	1.23	1.04	0	1.20	13	4.25	2.9	1.01
Max forks' length	2	4	3	0	16	1.47	17	10	1
Rewarded miners	20	29	74	0	404	556	807	43	0

Table 1: Summary of the most important configurations tried

When a miner chooses to compute the proof of work of a block, it needs to choose a number of transactions, which has a direct impact on the latency. To simulate it, we have introduced extra latency but also an extra reward for each transaction. A greedy miner could try to mine a huge block, with the only aim of making a big profit. The last column of Table 1 describes what we are

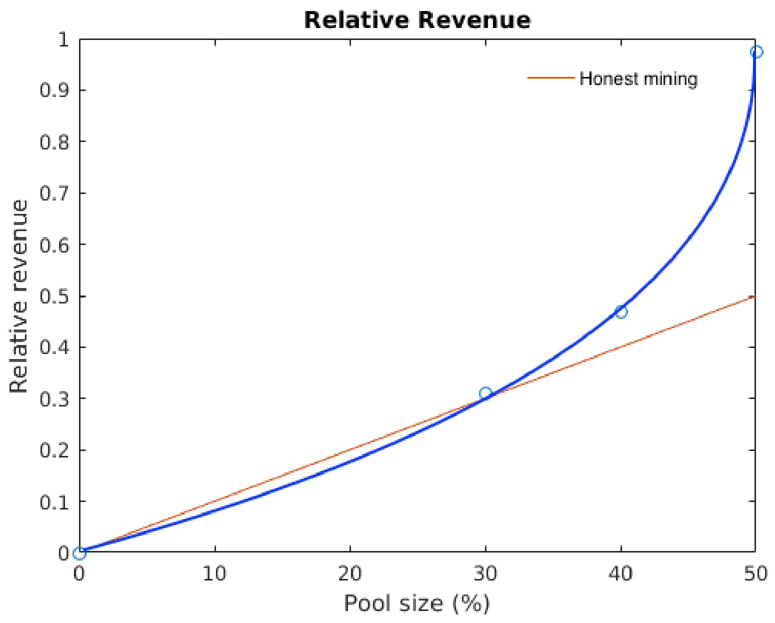

Figure 7: % of computational power of the pool vs relative revenue. The pool is realised as a single miner holding all the selfish power.

talking about. Since not all nodes may have a good network bandwidth, it is plausible to assume that a too large block will be spread into the network with a sensible delay. Each miner should then compute a trade-off between reward and time spend to spread the information, otherwise this can result into forks, and not only do they waste resources, but also cause the greedy node to get absolutely no reward in many cases. Thus, it should be safer to mine a block that would not take too much time to reach all the active nodes.

The last significant tests regard what happens when selfish miners join together in big pools. When latencies are negligible w.r.t a block spreading (as it should be in reality), we obtain the same curves as in [1], although the parameter γ has not been approximated but it seems to be near 0, given that the threshold over which this strategy makes sense is around 30%. The pool has been simulated assuming that there is just 1 selfish miner and it is chosen with high probability. The bigger the pool's computational power is, the longest are the forks, in that the pool can mine and hide blocks more frequently than the honest miners.

5 Conclusions

In this report we have analysed the selfish mining strategy. In order to do so, we implemented TinyCoin, a restricted version of Bitcoin which simplifies transactions and cryptography, as well as unpleasant details of dynamic networks. Results have shown that such dishonest miners should join in pools if they want to have a real advantage, otherwise they only obtain chaos and frequently waste their computational resources. Latency plays a similar role, although it is unlikely that it will ever be comparable to the block's mining time. Our simulator has been proved useful to confirm theories exposed in [1], and its implementation has been crucial in better understanding how Bitcoin works.

References

[1] Ittay Eyal and Emin Gün Sirer. Majority is not enough: Bitcoin mining is vulnerable. In *International conference on financial cryptography and data security*, pages 436–454. Springer, 2014.