Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing

Systems@ETH Zürich

Prof. T. Roscoe, Prof. R. Wattenhofer

Computer Systems

— Solution to Assignment 12 —

1 Eventual Consistency & Bitcoin

1.1 Delayed Bitcoin

a) It is true that naturally occurring forks of length l decrease exponentially with l, however this covers naturally occuring blockchain forks only. As there is no information how much calculation power exists in total, it is always possible a large blockchain fork exists. This may be the result of a network partition or an attacker secretly running a large mining operation.

This is a general problem with all "open-membership" consensus systems, where the number of existing consensus nodes is unknown and new nodes may join at any time. As it is always possible a much larger unknown part of the network exists, it is impossible to have strong consistency.

In the Bitcoin world an attack where an attacker is secretly mining a second blockchain to later revert many blocks is called a 51% attack, because it was thought necessary to have a majority of the mining power to do so. However later research showed that by using other weaknesses in Bitcoin it is possible to do such attacks already with about a third of the mining power.

b) The delay in this case prevents coins from completely vanishing in the case of a fork. Newly mined coins only exist in the fork containing the block that created them. In case of a blockchain fork the coins would disappear and transactions spending them would become invalid as well. It would therefore be possible to taint any number of transactions that are valid in one fork and not valid in another. Waiting for maturation ensures that it is very improbable that the coins will later disappear accidentially.

Note that this is however only a protection against someone accidentially sending you money that disappears with a discontinued fork. The same thing can still happen, if someone with evil intent double spends the same coins on the other side of the fork. You will not be able to replay a transaction of a discontinued fork on the new active chain if the old owner spent them in a different transaction in the meantime. To prevent theft by such an attacker you need to wait enough time to regard the chance of forks continuing to exist to be small enough. A common value used is about one hour after a transaction entered a block (∼6 blocks).

1.2 Double Spending

a) Figure 1 depicts the final situation. 7 nodes have seen T first and 5 nodes have seen T' first. The 5 nodes at the edge cut between the green and the red cut have seen both transactions.

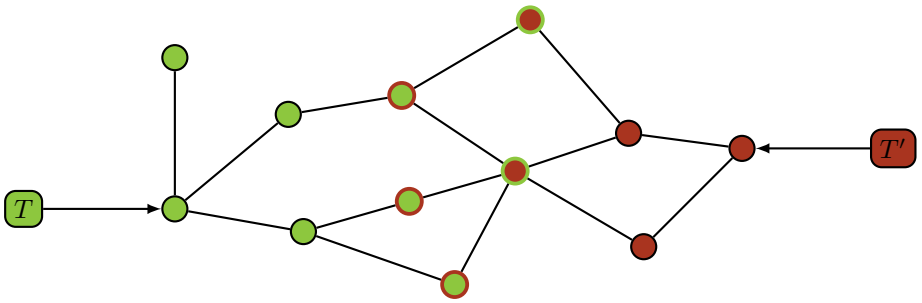

Figure 1: Random Bitcoin network

b) Each node has 1/12 of all computational resources, hence the probability of T being confirmed is $7/12 \approx 58\%$, while T' has a $5/12 \approx 42\%$ chance of being confirmed. The higher connectivity from the first node seeing T resulted in the transaction spreading faster, increasing the probability of winning the doublespend.

c) The first node that sees T' now has 20% of the computational resources. T' therefore has a probability to win of $2/10+1/11 \cdot 8/10 \cdot 4 \approx 49\%$. The distribution of computational resources in the network therefore matters. The goal of an attacker is to spread the transaction that she wants to have confirmed to a majority of the computational resources, which may not be the same as spreading it to a majority of nodes.

1.3 The Transaction Graph

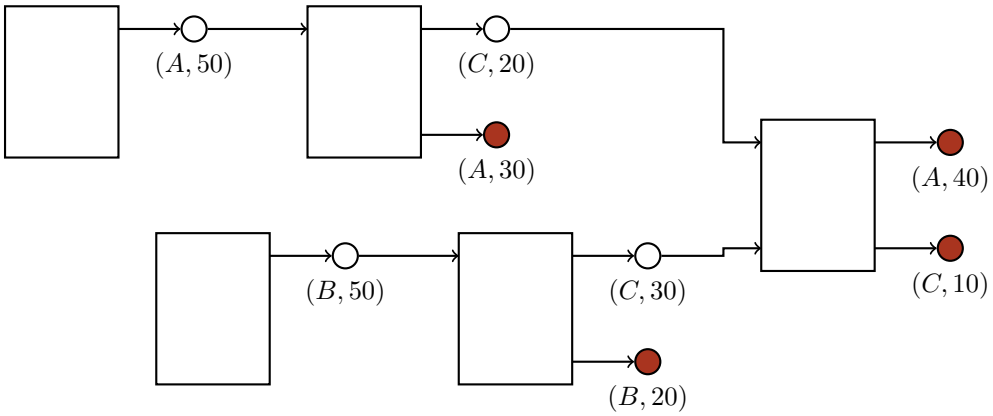

Figure 2: Transaction Graph. The red outputs are UTXOs.

a) See Figure 2.

b) See red outputs in Figure 2.

c) Fully spending an output simplifies the bookkeeping considerably as an output can only be in two possible states: spent or unspent. This means that it is easy to detect conflicts, because two transactions spending the same output are conflicts. If we were to partially spend outputs, allowing multiple transactions to spend the same output until the coins on that output were completely spent, then the conflicts become more complicated. Assume an output with value 1 bitcoin. When partially spending outputs we could create 3 transactions claiming 0.5 bitcoins from that output, two of them are valid and the third will be invalid,

but there are 3 possible combinations that are valid. So the simple answer is: it makes conflicts evident and reduces combinations for conflicts. The number of possible combinations increases rapidly with the number of transactions.

There are many points in the Bitcoin software where this would complicates things. A miner needs to construct a valid block from the known transactions, however this becomes more difficult if arbitrary combinations of transactions suddenly conflict with each other. Furthermore with complex conflicts attackers can create a situation where most nodes do not agree on the transactions which will be in the next block. However nodes are optimized to quickly be able to forward new blocks which look "expected". If the nodes do not agree at least loosely on the transactions to be committed in the next block, the propagation delays become much larger as many transactions need to be retransmitted, which finally results in smaller total transaction processing capability.

Note: As you might have realized the flow of money can be nicely followed with the transaction data from the Bitcoin blockchain. If some hacker steals your valuable coins, you can watch him buy things with it and see where the vendors are spending this money too! For this reason it is often possible to "buy" larger amounts of Bitcoin with less Bitcoin. The larger amount you can "buy" has very likely been involved in some crime and is being tracked by the police, thus the owner is eager to exchange them for other coins. Look for "Bitcoin doubling" in your favorite list of darknet services!

1.4 Bitcoin Script

a) Transactions are instantly finalized, so the large confirmation delay of the blockchain is irrelevant. Only the signatures of both parties are needed, then the money has effectively changed the owner. Furthermore no transaction fees have to be paid to miners for replacing a transaction.

b) Without the opening transaction A could just spend the money with a transaction without a timelock to a different address owned by himself. Requiring both signatures prevents this and gives security to B. In this construction B can trust that the funds will be available after the first timelock runs out.

Note that if B wants to access the funds earlier, it is still possible for A and B together to sign a transaction which directly executes the latest state. As long as both agree it is thus not necessary to wait for the timelocks. The timelocks are only necessary to ensure the last state in case there is disagreement.

c) A "kickoff" transaction can be introduced after the opening. Only the opening is executed (i.e., sent to the blockchain) at the beginning to secure the funds. Now transactions can be replaced and if someone wants to close the channel he can execute the kickoff. This starts the timers on the subsequent transactions. See Figure 3 for the new transactions.

In detail the protocol is the following.

Setup:

(a) A creates all transactions of the setup (opening, kickoff and first state).

(b) A sends these together with signatures for the kickoff and first state to B.

(c) B signs the kickoff and first state and sends the signatures to A.

(d) A signs the opening transaction and executes it on the blockchain.

Updating:

(a) A creates a new transaction spending the kickoff output with a lower timelock.

(b) A signs it and sends it to B.

(c) B sends his signature to A.

After the update both have a signed version of the new state and can terminate the channel in this state.

Closing:

(a) A or B proposes a settlement transaction that directly spends the locked-in funds in the opening output.

(b) The other party signs and sends it to the blockchain.

This is the cooperative closing case. If some dispute happens, either of the two parties can always send the kickoff and latest state to the blockchain.

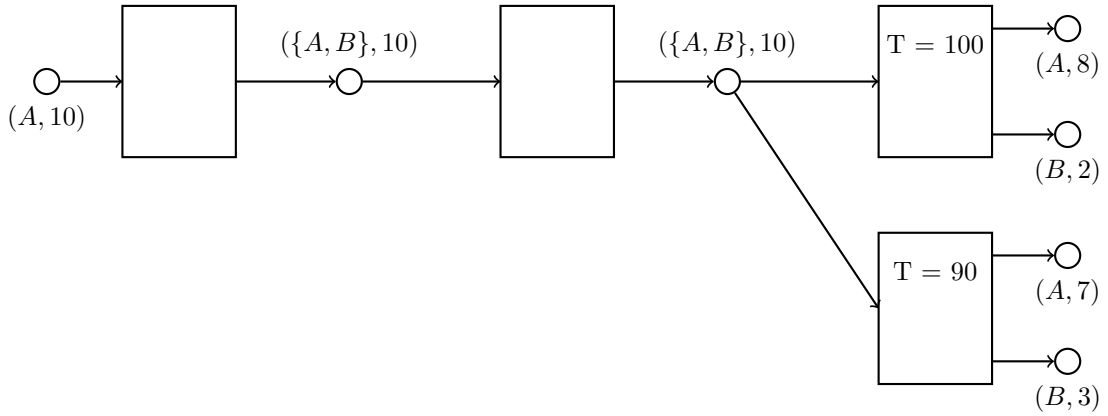

Figure 3: A "payment channel". A and B both have to sign to spend the output in the middle. The upper transaction can only be committed starting from blockheight 100, the lower one starting from blockheight 90.

2 Internet Computer

2.1 Consensus

a) With all nodes behaving honestly and a Δ-synchronous network, a block is guaranteed to be finalized in 5Δ time: First a block is reliably broadcast (3Δ). Exchanging notarization and finalization shares takes another Δ each.

A Byzantine node cannot delay finalization in this setting except if it is the rank-0 block maker. In which case it can delay proposing a block, in such a way that some nodes see the rank-1 block first. As no block will receive enough finalization shares, this can lead to no block being finalized in this round. Assuming the following rank-0 block maker is honest, the next round will start as soon as either the rank-0 or rank-1 block is notarized and $f + 1$ beacon shares are exchanged. The rank-1 block is broadcast 2Δ after the round starts. Its notarization will take 4Δ time, using the same argument as above. Exchanging the beacon shares takes an additional communication trip Δ. Thus, the next round will start 7Δ later. In total, a block will thus be finalized at the latest $7 + 5 = 13\Delta$ after the Byzantine-lead round started.

b) A canister message must first be broadcast to all nodes (Δ). Assuming the network is not under load, it will be included in the next block, the expected waiting time being $\frac{timeBetweenBlocks}{2}$. Thus, in the honest case the message will be included in a block after $\frac{4\Delta}{2} = 2\Delta$. Finally, the finalization of a block takes 5Δ if nodes behave honestly as seen above. Thus a canister message can be executed no faster than 8Δ after being broadcast.

c) With unbounded block size a Byzantine node could propose an extremely large block, bringing the network to a halt, both through longer transmission time, as well as longer execution time. Limiting the block size implies a cap on the possible throughput though. In turn, in periods of high utilization messages might not be included in blocks immediately, which means latency might increase.

d) The message needs to be available at all nodes to be executed, as otherwise, an adversary would be able to bring the blockchain to a halt by including hashes of non-existing messages. To avoid this the messages could for example be reliably broadcast before, and proof of successful broadcast can be included in the block (this can be done efficiently through BLS signature aggregation). Separating the dissemination of messages from the consensus layer leads to less communication overall, as a message is only transmitted once (instead of being first (best-effort)-broadcast to nodes and then re-broadcast inside blocks). Assuming that the bandwidth per node is bounded while computation speed is not, halving the communication complexity could mean doubling the throughput. Latency is increased by at least 2Δ, namely the time needed to collect enough signatures during the broadcasting phase.

e) The joining node needs to know the current nodes in the subnet and their peering details to connect to them. Further, it must receive its subnet private key share, the subnet state at a given height, all finalized blocks since that state, as well as all notarized blocks and their notarizations after the last finalized block. Finally, it must have the certificate of the last certified state and the most recent beacon shares in order to compute the nodes' ranks.

3 Advanced Blockchain

3.1 Selfish mining

Here's our (somewhat subjective and non-conclusive) answer to this question:
Until the next difficulty adjustment, the block generation rate with respect to the longest chain is reduced in the presence of a selfish miner as the total hashing power would be distributed on two forks of the chain. Here, the selfish miner might even get unlucky and make less profit than by following the protocol.

However, due to the difficulty adjustment, the rewards distributed among the miners finding blocks will be reestablished to match a constant rate. From the Selfish Mining Theorem in the script we know that it is rational (profitable) for a miner to mine selfishly if it has enough hashing power (e.g. $\alpha \geq \frac{1}{3}$ or even some smaller α with $\gamma > 0$).

Further thoughts:

- Assume that miners reinvest some of their income in buying new mining equipment. As the global rewards per hour are kept at a constant level due to the difficulty adjustment and a selfish miner would get disproportionally high rewards, the honest miners must be awarded disproportionally small rewards. This would mean that the selfish miner will have a greater increase in his hashing power in comparison to the honest miners. Consequently, his share of block rewards will increase even more.
- Assume that other miners are rational (miners trying to maximize their own profit). They would be better off joining the selfish miner. This poses the potential for a majority attack.

Remark: There are papers that compare selfish mining rewards vs. honest mining rewards over a longer term taking Bitcoin's difficulty adjustment into account. For example: `https://arxiv.org/abs/1912.01798` *and* `https://eprint.iacr.org/2018/1084.pdf`.

3.2 Smart Contracts

In the exercise description we already suggested some tools to achieve the given task, but of course other methods can be used. With tools like Ganache [1] you can simulate a blockchain locally and test your smart contracts without deploying. Development environments like Hardhat [2] can further help you with testing and deployment of your code.
In the following we list the changes that you have to make to the original contract to achieve the wanted additional functionality:

- Add a new mapping to check if a wallet that sent the transaction is allowed to mint:

```
mapping (address => bool) public additional_minters;
```

- Changing the require statement on line 23 to:

```
require(msg.sender == minter || additional_minters[msg.sender] == true);
```

- Add a new function to add additional minters:

[1] https://trufflesuite.com/ganache/
[2] https://hardhat.org/

```
function add_additional_minter(address additional_minter) public {
     require(msg.sender == minter);
     additional_minters[additional_minter] = true;
}
```

The above tools show that the actual building/deploying of smart contracts on Ethereum is quite straightforward. Smart contracts themselves can be complicated and need to be extensively reviewed/tested/formally verified. Many third party libraries provide standard functionality that is tested extensively beforehand. Most of the times, it's recommended to use these libraries. But sometimes, it can be quite catastrophic - see `https://www.parity.io/a-postmortem-on-the-parity-multi-sig-library-self-destruct/`.