COL106 : 2021-22 (Semester I)
Project

Sarang Chaudhari Venkata Koppula Anuj Rai Daman Deep Singh
Harish Yadav

November 1, 2021

Notations

For any natural number n, $[n]$ denotes the set $\{1, 2, \ldots, n\}$.

Important:

- Deadline: 06 November, 11:59pm
- Maximum score: 120

1 Introduction

In this document, we present our cryptocurrency called DSCoin. Recall, in Lab Module 5, you built a blockchain. Here, we will extend those ideas to have a (nearly complete) cryptocurrency. First, we introduce some basic rules/conventions/terminology:

- Every *coin* is a six digit unique number.
- Every *transaction* has the following information:
  - the coin being transferred
  - the source (that is, the person spending this coin)
  - the destination (that is, the person receiving this coin)
  - some information to indicate when the source received this coin from someone (this will be described in more detail later).

  For simplicity, we assume every transaction consists of exactly one coin.
- A *transaction-block* consists of a set of transactions. Let tr-count denote the number of transactions per block. [1] The transaction-block will also have additional attributes, which will be discussed below.
- A *blockchain* is an authenticated linked list of transaction-blocks.[2]
- *Pending transactions* and *transaction-queue*: All the transactions in the transaction-block are processed transactions. Additionally, we have a transaction-queue which contains pending transactions. Every new transaction is first added to the transaction-queue, and later moved to a transaction-block (and thus added to the blockchain).

[1] For ease of debugging, this will be a small number for our project. Real world blockchains have > 1000 transactions per block.

[2] Strictly speaking, this will not be a linked list; we will discuss this later.

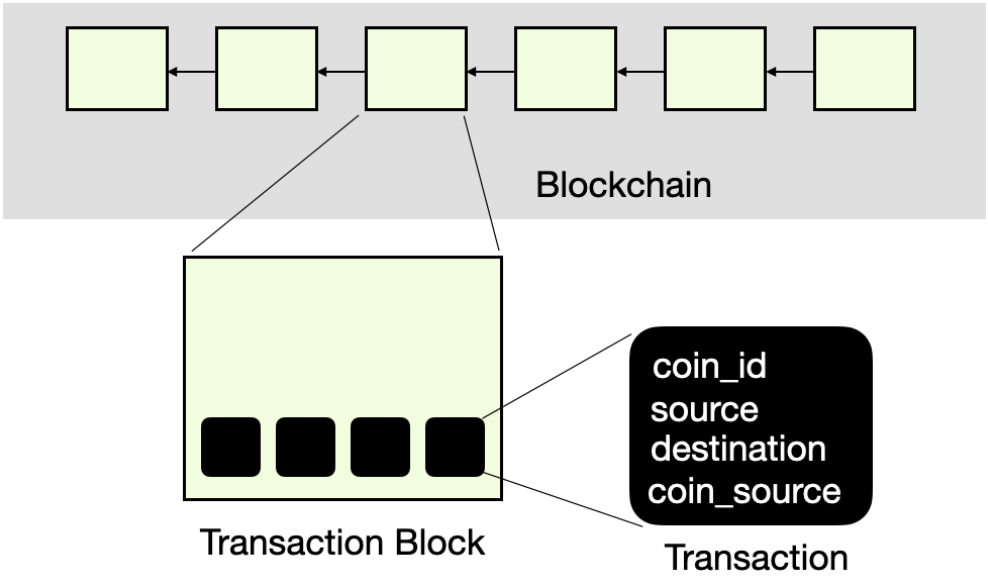

Figure 1: The blockchain is an authenticated list of transaction blocks. Each transaction block consists of transactions (together with additional attributes). Each transaction contains the coin-id, the source of the transaction (that is, the buyer), the destination (the seller) and coin-source (that is, the previous transaction block where the source received this coin).

2 DSCoin: Our Cryptocurrency

In our cryptocurrency, any participant can either be a

- *buyer* - someone who wishes to buy an item, and therefore send a coin,
- *seller* - someone who wishes to sell an item, and therefore wishes to receive a coin from the buyer
- *miner* - someone who verifies and approves the transactions

First, we will describe a cryptocurrency `DSCoin_Honest` where the miners behave honestly. In the next section, we will see how to handle malicious miners.

2.1 `DSCoin_Honest`: A cryptocoin for honest users

Initially, all participants receive a certain number of coins from the *moderator* (think of the IITD admin – this is the only role of the moderator). All these are added in (possibly separate) transaction-blocks to the blockchain and there are no pending transactions.

After this initial setup, suppose person A wishes to buy an item from person B, and therefore wishes to send one coin to person B (we will refer to person A as the *buyer* and person B as the *seller*). The buyer adds a transaction to the transaction-queue; this transaction must contain the coin-id he/she wishes to spend, the buyer's identification (say IITD entry number), and the seller's identification (again IITD entry number). Additionally, the transaction also *points to* a transaction-block in the blockchain where person A received this coin; we will call this the `coinsrc_block` of this coin. Therefore, any transaction is a 4-tuple : (coin-id, buyer-id, seller-id, `coinsrc_block`). This 4-tuple is added to the queue of pending transactions. Once there sufficiently many transactions in the transaction-queue, a *miner* removes them from this queue, and *mines* a transaction-block (we discuss how this is done in Section 2.1.2). After creating the transaction-block, the miner adds this block to the blockchain.[3] At this point, the buyer (person A) can check that his/her

[3]Note that anyone can be a miner, and therefore anyone can add a transaction-block to the blockchain. This should raise a red-flag: what if a malicious party is a miner, and adds invalid blocks to the blockchain? We will address this in Section 2.2.

transaction is included in the latest transaction block. If so, the buyer sends a *proof of payment* to the seller, and the seller can check this proof. To summarize, here are the steps involved in a transaction from a buyer to a seller:

1. Buyer adds transaction to the transaction-queue.
2. Miner collects many such transactions, mines a transaction-block and adds the block to the blockchain.
3. Once the block is added to the blockchain, the buyer checks that his/her transaction is present in the last block. If so, the buyer computes a 'proof of membership' of this transaction in one of the transaction block, and sends it to the seller.
4. Finally, the seller verifies this 'proof of membership'.

Below, we explain each of these steps in some detail.

2.1.1 Initializing a Coin-Send

Suppose buyer wants to send a coin to a seller. Every buyer/seller has a UID. The buyer creates a new transaction t, where the `coinID`, `Source`, `Destination` and `coinsrc_block` are set appropriately. After this, the buyer adds this transaction to the transaction queue. Additionally, the buyer also maintains his/her own queue of pending transactions (called `in_process_trans`), and the buyer adds the transaction to this queue.

2.1.2 Mining a Transaction Block: Honest setting

First, we discuss the structure of a transaction block, and then discuss how it is mined. A transaction-block consists of the following attributes:

- tr-count: the number of transactions in the block. We will assume this is a power of 2.
- `trarray`: an array of transactions.
- `trsummary`: a 64-character summary of the entire transaction-array. This is computed using a Merkle tree on the `trarray`. [4]
- `Tree`: the Merkle tree on `trarray`.
- `nonce`: a 10-digit string that is used to compute the `dgst`.
- `dgst`: a 64-character string, obtained by computing the CRF on previous digest, transaction-summary and a 10-digit string called nonce, separated by #. The nonce must be such that the first four characters of `dgst` are all zeroes.

The job of the miner is to create a transaction-block consisting of tr-count *valid* transactions (that is, none of the transaction should be a *double spending*). More formally, we say that a transaction t is invalid if any of the following holds true:

- the `coinsrc_block` of t does not contain any transaction t' such that t'.`coinID` = t.`coinID` and t.`Source` = t'.`Destination`.
- the above check passes, but this coin has been spent in one of the future transaction-blocks, or is present multiple times in the transaction queue.

The miner collects tr-count - 1 number of valid transactions from the `TransactionQueue`. The miner also gets a *reward* for mining this block. This reward, in our cryptocurrency, is one coin. The `coinID` of this coin will be the smallest six-digit coin-id (≥ 100000) that is available.[5] The miner creates a transaction with this `coinID`, sets the `Source` to be `null`, `Destination` is the miner, and `coinsrc_block` is set to `null`. This new coin belongs to the miner, and can be spent by the miner for future transactions. Therefore, the miner has tr-count number of transactions, which are included in the transaction block as follows:

[4] Strictly speaking, for each transaction t in `trarray`, we obtain a string by concatenating the attributes of t. The Merkle tree is computed on these strings.

[5] Ideally, this should be a random six digit number. For ease of testing, we opted for a deterministic approach - we will maintain the smallest six-digit number that is not used yet (stored as an attribute of the blockchain), and use that as the coin-id. The miner will also increment this attribute of the blockchain after mining the block.

1. adds the transactions to the transaction array `trarray`.
2. computes Merkle tree on the transaction array. Every node in the Merkle tree has a string attribute called `val`. For a leaf node corresponding to transaction, the `val` is simply CRF applied on a concatenation of the coin-id, source id, destination id and `dgst` of the `coinsrc_block` corresponding to this transaction (separated by #).[6] For any intermediate node, it is computed by applying the CRF on the concatenation of left child's `val` and right child's `val` (separated by #).
3. finds a 10-digit string `nonce` such that CRF applied on the previous block's digest,[7] the Merkle tree's root's `val` (i.e., also the current block's `trsummary`) and `nonce` (separated by #) outputs a string with first four characters being 0. This output is the new block's digest. Such a `nonce` can be found by sequentially searching over the space of all 10-digit strings.

At this point, the miner has computed all the attributes for the new transaction block. The miner simply adds this transaction block to the blockchain.

2.1.3 Finalizing a Coin-Send

Once a transaction t is added to a transaction block in the blockchain, the buyer can convince a seller that he/she has sent a coin to the seller. Suppose the transaction is included in a block b_0, and there are r blocks $b_1, b_2, \ldots, b_r$ in the chain after b_0 (that is, b_r is the last block in the blockchain, b_{r-1} the previous block, and so on). The buyer first computes a sibling-coupled path to the root of b_0's Merkle tree. The buyer sends the following information to the seller:

- the sibling-coupled-path-to-root corresponding to transaction t
- the `dgst` of b_0.`previous`.
- the `dgst`, `nonce` and `trsummary` for each b_0, $b_1, \ldots, b_k$.

We will refer to this entire tuple as the *proof of transaction*. The buyer also removes this transaction from his/her own `in_process_trans` queue.

2.1.4 Verification of Transaction by the Seller

Suppose a seller is the destination for a transaction t in block b_0, and suppose there are k blocks $b_1, \ldots, b_k$ in the blockchain after b_0. The seller receives a sibling-coupled-path-to-root, the `dgst` of b_0.`previous`, the $(\texttt{dgst}_i, \texttt{nonce}_i, \texttt{trsummary}_i)$ pairs for each of the blocks b_i, $i \in [0, k]$. The seller checks the following:

- first checks that he/she is indeed the destination for the transaction t.
- next checks the sibling-coupled-path-to-root, and checks that the final value in the sibling-coupled-path-to-root is equal to $\texttt{trsummary}_0$.
- checks that each of the $\texttt{dgst}_i$ strings are valid (for $i \in [0, k]$) – the seller checks that the first four characters of $\texttt{dgst}_i$ are 0, and that $\texttt{dgst}_i$ is correctly computed using $\texttt{dgst}_{i-1}$, $\texttt{nonce}_i$ and $\texttt{trsummary}_i$.[8]
- finally checks that $\texttt{dgst}_k$ matches the `dgst` of the last block in the blockchain.

2.2 `DSCoin_Malicious`: Handling Malicious Miners

The solution described in the previous section works in the setting where all miners are honest. [9] However, one of the prime advantages of a cryptocurrency is the decentralized aspect, and in this setting, we cannot assume that the miners are honest. Indeed, it is possible that a buyer (source) adds an invalid transaction to the `pendingTransactions` queue, and then the same buyer mines a block that includes the invalid

[6] If any of these values are `null`, then you should use the string "*Genesis*".
[7] If this is the first block, then you should use the start string instead.
[8] For $i = 0$, $\texttt{dgst}_{i-1}$ is the `dgst` of b_0.`previous`.
[9] For instance, if the miners were appointed by the moderator, and being a miner is a *Position of Responsibility*, then we can assume that all miners are honest.

transaction. Or maybe the malicious buyer has a miner friend who includes the invalid transaction in the transaction block.

Interestingly, malicious miners are handled using a clever mix of incentive-engineering and data structures (and no additional cryptography involved here). First, the blockchain is generalized – instead of having a linked list, we have a tree-like structure, where we store all *leaf blocks* of the tree, and for each block, we have a pointer to the `previous` block as before. For instance, in Figure 2, we have three leaf blocks – blocks numbered 9, 11, 12. Note that anyone can identify these maliciously mined blocks – one only needs to check the validity of each transaction in the block, check if the Merkle tree is computed correctly and finally check that the digest is computed correctly. We define a transaction block to be *valid* if the `dgst` is correct, all transactions in the block are valid and the Merkle tree is computed properly. You can assume that the first few blocks (created by the moderator) are valid blocks. The *longest valid chain* is defined to be the longest sequence of blocks, starting with the first block, consisting of only valid blocks. If a honest miner is mining a new block, then the miner finds the longest valid chain, and attaches the new transaction to this valid chain. For instance, in Figure 2, the longest valid chain is the sequence $(1, 2, 3, 4, 6)$, and therefore, the new block should have 6 as its previous block. If 6 was also an invalid block, then the longest valid chain terminates at 4. Finally, if 5 was a valid block, then the longest valid chain would be terminating at 12. [10]

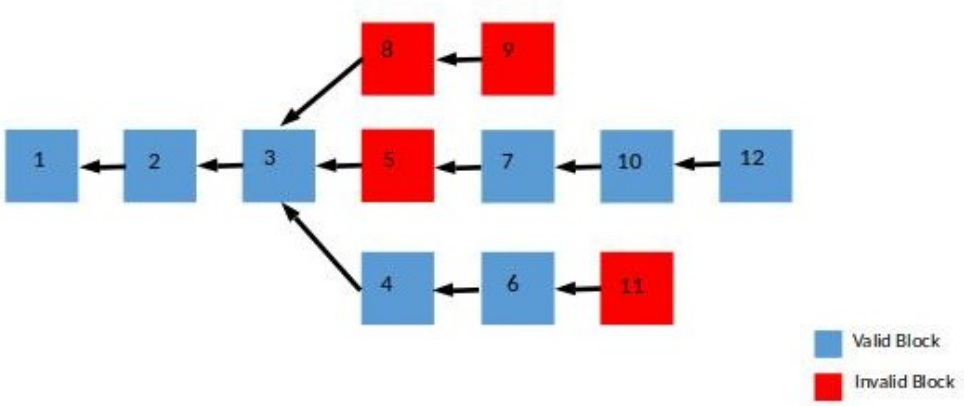

Figure 2: The malicious blockchain has a tree-like structure. Each block has a pointer to the previous block, and we store all leaf-blocks. In this figure, there are three leaf blocks, and four invalid blocks (shown in red).

2.2.1 How Does the Above Deter Malicious Mining

If the majority of miners are honest, then eventually, the longest chain in the blockchain will consist of only valid transactions, and malicious blocks are thus *discarded*. You can assume that the cryptocurrency software automatically checks that the digest is correctly computed, and has an appropriate number of zeroes. But the other validity check – ensuring that there is no double spending – is done by the fellow miners.

The role of the nonce, and how it ensures that there are no invalid transactions in the blockchain: In this assignment, we want the first four characters of the `dgst` to be zeroes, but in real-world cryptocurrencies, many more initial characters need to be zeroes.[11] Therefore, computing the nonce is the most expensive operation in the computation of a new block, and requires a lot of computational resources. The reward coin given to the miner is the incentive for the miner to behave honestly. If a miner has invested so much computational resources in computing the correct `nonce` and `dgst`, then any reasonable miner will

[10] For the test cases in this assignment, you can assume that there will be a unique longest valid chain.

[11] In cryptocurrencies, one does not need to output the smallest nonce; any nonce is allowed. However, finding *any* nonce seems to be as hard as finding the smallest nonce.

also ensure that the transactions in the transaction block are valid (otherwise, if the miner's block is invalid, then all the computing effort goes waste).

2.3 Conclusion

The cryptocurrency described above is close to real-world cryptocurrencies (such as Bitcoin). Here are some differences:

- Here, we assumed every person has a UID (the kerberosID). In Bitcoin, every person chooses a signing/verification key, and the verification key is the person's UID. Whenever someone mines a block, he/she also computes a signature on the block. [12]
- Here, we assumed that every transaction consists of exactly one coin spending, and every miner receives one coin as reward. In real life, a transaction allows person A to send any number of coins to person B. Moreover, person A can also include a 'reward' for the miner. This incentivises the miner to include person A's transaction in the transaction block. In this scenario, how should you implement the pending transactions queue? And how would you modify the mining procedure?
- Bitcoin uses a cool stack-based scripting language. Every transaction includes a small program in this programming language. You can read more about it here.

Finally, if you are interested in learning more about cryptocurrencies, check out this excellent book.

[12] Earlier, we intended to include this in the project, and hence discussed signatures in LM1. However, given that the project has enough components already, we skipped this part.