CMPE 476 – Assignment 4 @ CmpECoin: Blockchain Implementation

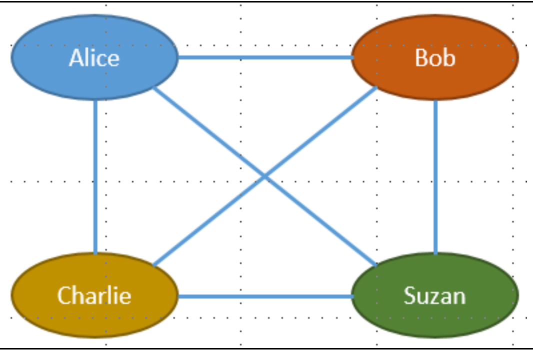
Due Date: 2021.06.16 (June 16) @23:59

Ledger – (no need of) Trust + Cryptography = Cryptocurrency

In this project you are required to implement a simple blockchain. There will be a demo session for your project after the submission.

1. Ledgers and Digital Signatures

Alice, Bob, Charlie, and Suzan are very close friends with a strong trust relationship. Suppose that, from time to time Alice, Bob, Charlie, and Suzan pays for each other depending on their cash and suppose that they keep a ledger for their transactions as follows:

Transaction Ledger: Alice pays Bob 40 TL Bob pays Charlie 90 TL Suzan pays Bob 30 TL Charlie pays Suzan 50 TL	

Protocol Among Parties v01

- Anyone can add lines to Ledger
- Settle up with real money each month

Clearly, this protocol depends on complete trust among the parties. There is no mechanism that stops Bob adding a line like "Bob pays Alice 100 TL". One mechanism can be putting a signature next to the corresponding line.

If the ledger is stored digitally, we should have digital signatures with the following properties:

- Signatures should prove the signer's identity
- It should not be feasible to forge a digital signature
- It should not be possible to copy the signature from a previous document

These properties are satisfied by use of a public key (pk) and a private/secret key (sk) pairs. As the name suggests, you keep sk hidden from everyone. In "analog" life, your signature looks same whatever document you sign. However, digital signatures change from document to document.

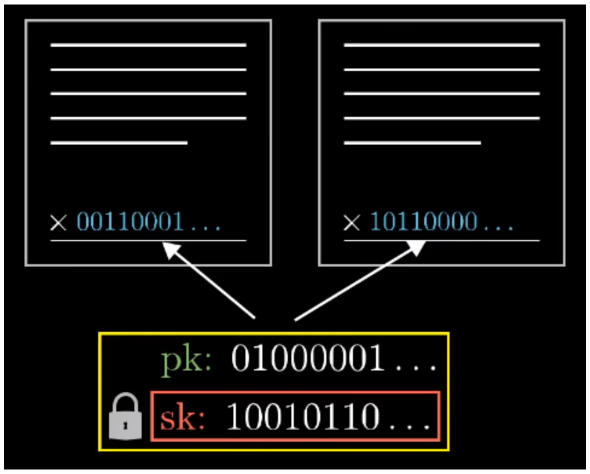


Figure 1- Two different documents are signed with a secret key and the signatures are shown at the bottom of the document. Since the signatures also depend on the document they are most probably different from document to document. Altering the message slightly yields to a different signature

So the signing and verification functions works as follows:

```
Sign(Message, sk) = 256-bit-Signature

Verify(Message, 256-bit-Signature, pk) = True/False
```

The number of possible 256-bit-Signature values is very huge so that if the Verify() function returns True, then you are very confident that the message is signed with the corresponding secret key (**sk**).

In the light of these, we are sure that we need to add a signature to verify the identity of the line adding person. However, we also need to put a unique value to the message (i.e., transaction line), otherwise, anyone can copy and paste the same line multiple times.

Now we are building a better and more secure system and the new protocol becomes:

Protocol Among Parties v02

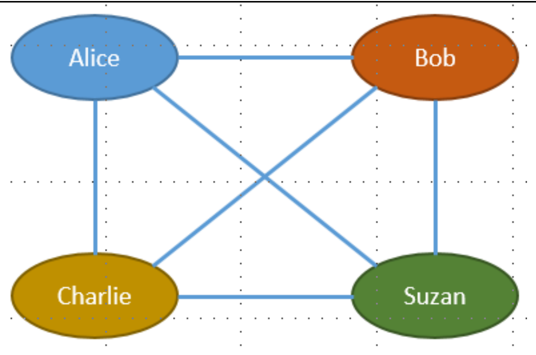
- Everyone puts some money to the system and no overspending will be allowed
- Anyone can add lines to Ledger
- Only signed transactions are valid

Now consider the following scenario

Transaction Ledger:

1. Alice gets 100 TL (Init)
2. Bob gets 100 TL (Init)
3. Charlie gets 100 TL (Init)
4. Suzan gets 100 TL (Init)
5. Alice pays Bob 50 TL <256-bit-Signature>
6. Alice pays Suzan 50 TL <256-bit-Signature>
7. Alice pays Charlie 50 TL <256-bit-Signature>

INVALID- After 6 Alice's balance is 0 TL



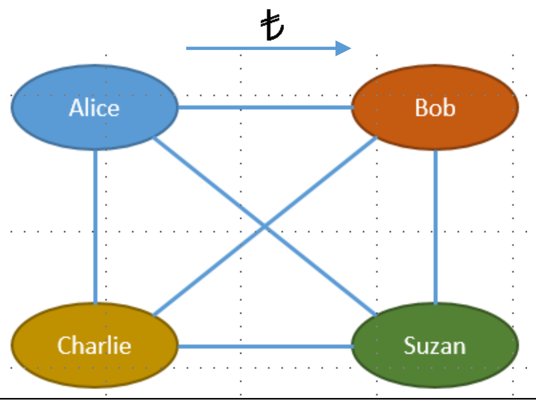
So verifying a transaction requires

- knowing the full history of transactions up to that point
- verifying the digital signature

From now on, let's change our currency type in ledger to CmpECoin (CEC). In real life we may exchange TL banknotes for an exchange of CEC. Suppose that 1 TL is currently 1 CEC and Alice gives 20 TL to Bob for a transfer of 20 CEC to herself and in that case ledger will have the following line:

Transaction Ledger:

…
…
…
6. Bob pays Alice 20 CEC <256-bit-Signature>



So at this point we are ready to give the definition of a cryptocurrency.

Definition: A ***cryptocurrency*** is a digital asset designed to work as a medium of exchange wherein individual coin ownership records are stored in a ledger existing in a form of a computerized database using strong cryptography to secure transaction records, to control the creation of additional coins, and to verify the transfer of coin ownership [wikipedia].

So the building block of a cryptocurrency is its ledger containing the whole transactions.

However, we are not there yet. The next question should be: "Who hosts the Ledger? Who controls it?"

2. Decentralizing the Ledger

Suppose that the nodes are connected via a network and the nodes are broadcasting the transactions over this network. Also suppose that every node stores the ledger by themselves. But, here is the most critical question: “How can you be sure that every node that is listening the broadcasted transactions is recording the same transactions in the same order?”. This is the core issue in cryptocurrencies. Can we come up with a protocol for how to accept and reject transactions and in what order so that you can feel confident that anyone in the world who is following the same protocol has a local ledger that looks the same as others. This is the original problem that is addressed in the BitCoin paper.

Protocol Among Parties v03

- Broadcast transactions
- Only signed transactions are valid
- No overspending
- What to add here for decentralizing and agreeing on how to accept and reject transactions

2.1. Cryptographic Hash Function

Definition: A ***cryptographic hash function*** is a mathematical algorithm/function that maps data of arbitrary size (often called the "message") to a bit array of a fixed size (the "hash value", "hash", or "message digest"). It is a one-way function, that is, a function which is practically infeasible to invert [wikipedia]. Ideally, the only way to find a message that produces a given hash is to attempt a brute-force search of possible inputs to see if they produce a match, or use a rainbow table of matched hashes.

The ideal cryptographic hash function has the following main properties:

- it is deterministic, meaning that the same message always results in the same hash
- it is quick to compute the hash value for any given message
- it is infeasible to generate a message that yields a given hash value (i.e. to reverse the process that generated the given hash value)
- it is infeasible to find two different messages with the same hash value
- a small change to a message should change the hash value so extensively that a new hash value appears uncorrelated with the old hash value (avalanche effect)

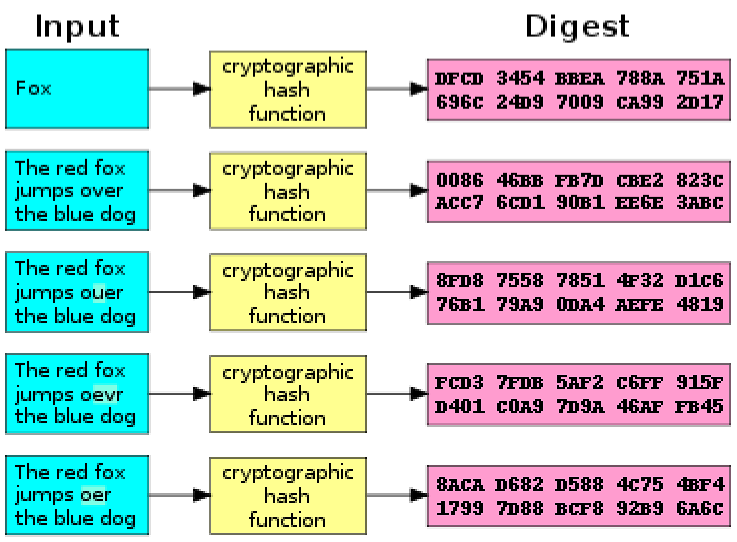

Input
Digest
Fox
cryptographic hash function
DFCD 3454 BBEA 788A 751A 696C 24D9 7009 CA99 2D17
The red fox jumps over the blue dog
0086 46BB FB7D CBE2 823C ACC7 6CD1 90B1 EE6E 3ABC
The red fox jumps ouer the blue dog
8FD8 7558 7851 4F32 D1C6 76B1 79A9 0DA4 AEFE 4819
The red fox jumps oevr the blue dog
FCD3 7FDB 5AF2 C6FF 915F D401 C0A9 7D9A 46AF FB45
The red fox jumps oer the blue dog
8ACA D682 D588 4C75 4BF4 1799 7D88 BCF8 92B9 6A6C

Figure 2- A cryptographic hash function (specifically SHA-1) at work. A small change in the input (in the word "over") drastically changes the output (digest). This is the so-called avalanche effect. [wikipedia]

Cryptographic hash functions will enable the concept named "Proof of Work" that is used for agreeing on the transactions.

2.2. Proof of Work (PoW)

From now on, we will consider a set of transactions as blocks. Now suppose that we are putting a number under a block (i.e., set of transactions). How hard can it be to find a number that makes the SHA256 hash/digest of a block starting with 30 zeros.

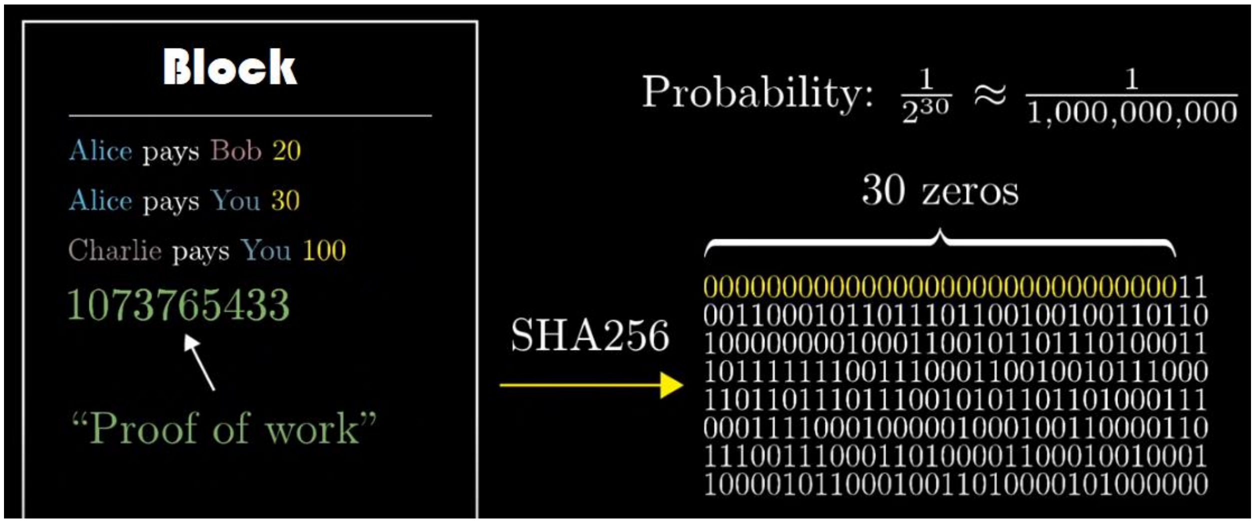


Figure 3- Here Proof of Work is found by brute-force and any change in the transactions will require new brute-force to find the appropriate number that yields 30 zeros at the beginning of the digest. This number i.e., the PoW is intrinsically tied to the transactions in the block.

2.3. Distributed Ledger – Blockchain

Now let the nodes broadcast their transactions via a network and let the set of transactions, i.e., the blocks are listened and added to the chain of blocks as follows:

- Each block consists of transactions that are signed digitally by the broadcaster
- Each block has a PoW value with difficulty level D, where D is the number of zeros at the beginning of the digest
- Each block contains the hash/digest value of the previous block

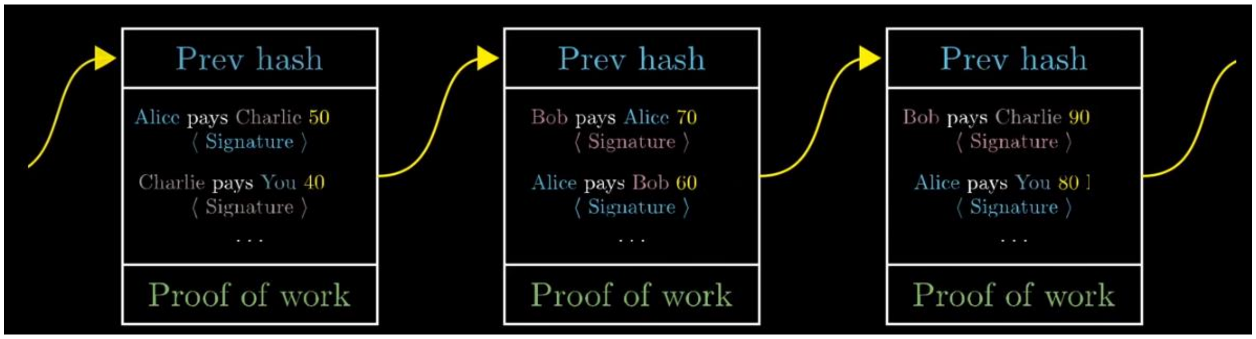


Figure 4- Blockchain structure. If a block is altered or the order is changed, then the whole Proof of Work must be done again

In this architecture, there are block creators/miners, whose job is to listen for the transaction in the network and put them in a block and find the PoW value for that block. Block creators are in a race for finding the PoW value since they are allowed to put a special transaction on top of the block (***of course before finding the PoW***) to transfer a reward to themselves that is created from thin air. If somehow two block creators find the PoW for the block concurrently, the network will stick to the longest chain and in the long run the more valid chain will stay alive. If a node tries to change a block and create all the PoW up to latest block, it needs to race with all the other block creators working collaboratively on the most valid chain. By luck, the node with the aim of altering the blockchain may find PoW values faster than the other nodes for few blocks, but after some point the original chain will win since more nodes are working on it. So it is really hard to alter the blockchain, it requires all the PoW values to be recalculated and in faster way compared to the other block creators, which may require the control of at least 51% of the block creators.

From block creators’ perspective, each block is a kind of lottery, they randomly try to find the PoW value for the reward coin. Every node in the coin-network can become a block creator, all they need to do is to listen for the transactions in the network and work on the block creation. After PoW value is found, the created block (with PoW value embedded) is broadcasted to the whole network. If a node in the coin-network hears two different chains, node needs to stick to the longest one that has more proof of work in it. If there is a tie, just wait for another block until the tie is resolved.

By doing so, you replace the need of trusting to a central authority with the trusting to the computational work. This protocol enables to reach a decentralized consensus.

3. Implementation of CmpECoin Network

This section gives hints about the implementation task and explains some requirements. In this project, you are required to implement a very simple cryptocurrency network where the nodes are processes or threads. For the communication protocol, you may use sockets, message queues, or a similar system. The explanations are given according to message queues.

Although it is possible to have a design without a central node, we will have central dispatcher node for simplicity. Dispatcher node will have very simple duties to focus on other aspects of the implementation.

- we will use a communication dispatcher, all of the communication will be carried over a special node with a specific/well known address. Transactions will be sent to dispatcher and the dispatcher will broadcast it to the network nodes. Validator nodes will send the validated block to the dispatcher node and the dispatcher node will broadcast it to the network nodes if it is the earliest “next” block with the appropriate PoW value. Dispatcher will not broadcast a conflicting and an earlier block. Some of the validators may find PoW and send the validated block to the dispatcher a bit late and they may not be aware of the situation. Dispatcher should be checking these kind of situations before broadcasting a block to be added to the chain. So the synchronization issues will be handled by dispatcher.
- dispatcher will also send beacons every 5 seconds for indicating the block validation start times.

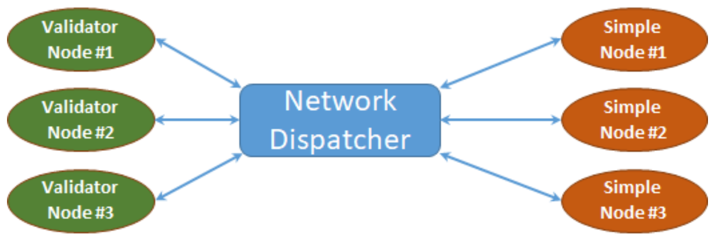


Figure 5- Network dispatcher and the communication model of the nodes in the topology

3.1. Skeleton of The Classes

Here, the baseline for the classes will be introduced. It is not a complete set of classes and the fields and the functions can be changed according to needs arising during the implementation. These classes and their fields and functions should give a hint about the structure of the system and its parts.

CmpECoinNetwDispatcher
transxRcvQ
validatedBlockRcvQ
joinedNodeAddresses
--- 	
broadcastReceivedTransx()
broadcastLastValidatedBlock()
broadcastValidationBeacon()
sendCurrentBlockChainTo(address)
sendSimpleNodeAddresses()
	This Class stands for the network dispatcher and it has two important MessageQueue instances (transxRcvQ and validatedBlockRcvQ) for listening the transactions sent by “Simple Nodes” and the validated Blocks sent by the “Validator Nodes”. Main duties of network dispatcher are
	• listening transactions from simple nodes and broadcasting them to validator nodes
	• sending beacon to indicate the start of the validation slot
	• listening validated block(s) for the current validation slot, check the validated block(s), decide the first validator, and broadcast the winner validated block. During this time simple nodes should be able to do new transactions.
	• handling the join requests. The current block chain should be sent to newly joining node

CmpECoinValidatorNode netwDispatcherAddress blockChain:BlockChain listenQForTransactionsFromNetwDispatcher listenQForValidatedBlocksFromNetwDispatcher --- joinCmpECoinNetw() handleReceivedTransactions() handleReceivedValidatedBlock() handleBeaconAndStartValidationProc()	This Class stands for the “Validator Node”. It has two important MessageQueue instances for listening transactions and validated blocks broadcasted by network dispatcher. Transactions are put into the pending transactions list/vector until the beacon arrives for indicating the start of the validation slot. When the beacon arrives, validator node puts a reward for itself and it tries to validate the block by finding the PoW value by trying randomly. Via the othe MessageQueue instance, validated and approved blocks are received from dispatcher. These blocks are added to the blockchain. **These nodes should also have a wallet** for collecting rewards.
CmpECoinSimpleNode netwDispatcherAddress blockChain:BlockChain myWallet meanTransactionInterDuration meanTransactionAmount listenQForValidatedBlocksFromNetwDispatcher --- joinCmpECoinNetw() handleReceivedValidatedBlock() doRandomTransactions() doRandomInvalidTransaction()	This Class stands for the “Simple Node”. Instances of this class basically do random transactions to other simple nodes and keeps track of the current state of the blockchain and their wallet. Simple nodes should generate an exponentially distributed random duration with the indicated mean and they wait that random amount of seconds and do a random transaction by their own CmpECoin in their wallet. Do not forget to sign the transactions because validator nodes will check the signatures for the transactions that they hear.

CmpECoinWallet
publicKey
privateKey
currentBalance
--- 	
initWallet()
getPublicKey()
getPrivateKey()
getCurrentBalance()
setCurrentBalance()
	It is a simple Class for holding the information about cryptocurrency wallet.

CmpEBlockchain
chain
pendingTransactions
validationReward
difficulty
--- 	
createInitialDummyBlock()
isChainValid()
addTransactionToPendingList(transx)
validatePendingTransactions(rewardAddress)
getBalanceOf(address)
getAllTransactionsFor(address)
	isChainValid() function Loops over all the blocks in the chain and verify if they are properly linked together and nobody has tampered with the hashes. By checking the blocks it also verifies the (signed) transactions inside of them.
	
	validatePendingTransactions() function takes all the pending transactions, puts them in a Block and starts the validation process. It also adds a transaction to send the mining reward to the given address.

CmpEBlock prevBlockHash currBlockHash timestamp transactions proofOfWork --- calculateCurrBlockHash() validateBlock(difficulty) hasValidTransactions()	validateBlock() function starts the mining process on the block. It changes the 'proofOfWork' **randomly** until the hash of the current block starts with enough zeros determined by *difficulty* value hasValidTransactions() function validates all the transactions inside this block (signature + hash) and returns true if everything is okay.
CmpETransaction fromAddress toAddress amount timestamp signature --- calculateTransactionHash() signTransaction(secretKey) isTransactionValid()	signTransaction(secretKey) function signs the transaction with the given secretKey. The signature is then stored inside the transaction object and later stored on the blockchain.

3.2. Demo Scenario

There should be 3 simple nodes, 3 validator nodes, and a dispatcher node. When the nodes are initiated, they should join the network via communicating with the dispatcher node and after that point, all the communication will be carried over the dispatcher node.

You should visualize the actions of the simple and validator nodes and their e-wallets. It should be possible to give a command to simple nodes for creating an invalid transaction and broadcasting it to the dispatcher node with hoping to be validated.