Final Term PeerToPeer&Blockchains (2019-2020)

Giulio Purgatorio, 516292

I've pasted the questions and then proceeded to answer them directly in order to avoid confusion.

1. Analysis of the Bitcoin protocol

Use the https://blockchain.info/ web site to interactively browse the Bitcoin blockchain and to solve the following questions.

1.a

Examine block number 3518846 in the blockchain. (https://www.blockchain.com/btc/block/351846)

- **How much in total did the miner who found this block receive for doing so?**

  The miner received a total of **25.00946136 BTC** for finding the block.

- **Locate the second transaction in this block (the one with transaction id 18b37c44...). How many inputs and outputs are there in this transaction?**

  In the transaction identified by 18b37c44… there are **1 input and 2 outputs**.

- **Give a likely explanation for why the two recipients do not receive the same amount.**

  5.5 BTC is the payment to someone, **0.5 BTC** is the change of the transaction that is put in another wallet. It may or may not be of the same owner, we can only see by inspecting the address that he got two transactions that immediately spent for others (both in less than a day. Reference link: https://www.blockchain.com/it/btc/address/14nvztZvmosVEsPRdwAcH5snmVRPopENms). An easy explanation would be that, assuming it's not the same owner, it's a debt to someone that got paid like that. I think this is highly unlikely just because of the "fast spending", so it could be the same user.

  I. **To 6 decimal places, what is the total sum of the input for this transaction?**

     6.086934 BTC.

  II. **What is the total sum of the output of this transaction?**

     6.08673438 BTC. *(5.57673438 + 0.51000000)*

  III. **Why is there a difference?**

     The difference is due to the **fee**, exactly for: 0.0002 BTC.

  IV. **What are the first 6 characters of the recipient who received the difference between the input and output for this transaction?**

     1LH3Qt

Examine block number 351833 in the blockchain. (https://www.blockchain.com/btc/block/351833)

- **What is odd about this block? Why do you think this occurred?**

It's a **Coinbase** transaction. Generate BTCs to reward miners for their work, and are characterized by having 0 inputs and 1 output. This output is the miner's address, to transfer BTCs to them.

- **Who owns the output address of the transaction with id 6848c…..?**

AntPool, which is the miner of the block.

- **Look at all of the transactions that the miner who found that block has received. Roughly how much USD is this?**

Nearly 1.250.000.000 USD *(based on the value of BTC at 21/05/2020, 16:33)*

- **How has this miner managed to win so many blocks?**

To win this many blocks, it must have a lot of computational power. Usually, this means that he worked in a group, combining its computational power with other miners: this is called a **"Mining Pool".**

- **Describe how the miner has likely managed the rewards it received for all these transactions.**

Assuming that he's in a Mining Pool, the rewards must be split among the members. The winner will of course get the most of it, but others must be rewarded as well, otherwise the Mining Pool would make no sense at all. There are many ways to handle this, the most used one is being awarded for "close enough blocks" that weren't successful but still prove the fact that a miner actually worked towards the resolution. Other solutions may depend on the computational power of a member, or simply share them equally (through a centralized mining operator which the miners will trust or through a private blockchain).

- **How "hard" was the block to find? On average, how many nonces would the miners executing the PoW contemporary to the winning miner had to try before finding a satisfactory nonce?**

The difficulty is: 49.446.390.688,24
(from: https://en.bitcoin.it/wiki/Difficulty):

*time = difficulty * 2^32 / hashrate*

-> *hashrate = difficulty * 2^32 / time*

In the Bitcoin case, the difficulty is adjusted every 2016 blocks (nearly 2 weeks) to reach an average of 10 minutes.
I've calculated with this command:

*python -c "print (49446390688*2**32/600)"*

nearly 354 petahashes per second.

So, how "hard" was the block to find isn't easy: if we take into account modern blocks, it was easy. https://www.blockchain.com/btc/block/581846 is a block mined nearly last year and it has a difficulty of

7.409.399.249.090,25, which is around 150 times the other block. Furthermore, hashrate power increases over time, as the Moore Law states. Anyway, it has been definitely a tough task.
To have an idea of today's standards, there's an estimation (2 Jan 2020) that BTC's hashrate is around 120 Exahash.

1.b

Answer the following questions.

- **Name three typical spending conditions that are supported in Bitcoin.**

Pay-to-Public-Key
The simplest one. When a transaction is inserted in the network, each node takes the entire script and executes it. Only one operator CHECKSIGs. An important thing to note is that locking and unlocking are done in two different transactions.
When a node of the P2P network receives a transaction, it will have to validate it. It takes the unlocking code and the reference to the previous output that has locking code in it. Concatenate the two scripts and run them together.

Pay-to-Public-Key-Hash
The most popular script. The lock contains the hash of a public key, instead of the public key itself. The unlocking script must provide a public key that returns the hash of the address contained in the locking script and a signature corresponding to the public key. The script execution hashes the public key and compares it to the hash in the locking script and then checks if the signature is valid.

Pay-to-Multisignature (or simply, Multisignature)
A script that locks bitcoins to multiple public keys and requires signatures for a set of these to unlock them. Common ways are:

- **1-of-n** anyone of n different parties can approve the transaction.

Example: one of three employees can spend some funds and the blockchain must contain a record of who did so.

- **2-of-2** each of two separate parties must approve the transaction.

Example: two departments in a company must sign off before some data is published on a blockchain, on behalf of that company.

- **2-of-3** any two out of three parties can approve the transaction.

Example: commonly used for escrow contracts, the two counterparties to an agreement engage a third party to act as an arbitrator in the event of a dispute.

- **Explain the three-step protocol used for exchanging new transactions/blocks in the Bitcoin P2P network.**

  The first step is the transaction **creation**. The second step is the **forwarding** of the transaction, which is done by broadcasting in the BTC network: it'll eventually reach all nodes. Finally, for the third and last step, the **validation** through the insertion of the transaction in a block that will be validated by a miner. All these validations will take some time, because of possible forks: a good measure is that if a block has 6 sequent blocks, it's both validated and secure.

- **List at least 3 differences between Bitcoin's UTXO model and Ethereum's account-based model.**

  First of all, accounts in Ethereum are in a Database that maintains the status and transactions change this status. So **UTXO** is **stateless**, while **Ethereum** accounts are **not**. About this, Bitcoin only has "simple" accounts, while in Ethereum there are EOAs (Externally Owned Account), used only for transferring Ether to and from, and smart-contract accounts, used also in order to implement a desired functionality.

  Then the **supply** is **limited** in **Bitcoin** (up to 21M), while in **Ethereum** it's **infinite**: this is possible due to the inflationary monetary policy of Ethereum, opposed to the deflationary Bitcoin's one. This also leads to an important difference speaking about the mining reward, wherein Ethereum is constant and in Bitcoin halves every 210000 blocks (so nearly 4 years).

  Another difference is that Ethereum has ***almost-Turing-complete*** smart contracts (written in Solidity), differently from Bitcoin scripts. The *"almost"* depends on the *Gas*, which simply denies all possible infinite loops.

  Furthermore, Ethereum has a very fast **mining rate** (nearly 10-20 seconds) compared to the BTC one which averages at 10 minutes.

  Finally, Bitcoin is based only on the **Proof-Of-Work** (PoW), while Ethereum is moving towards the **Proof-Of-Stake** (see Casper, removing the mining process from Ethereum).

- **What information is required to prove that a transaction is in a block?**

  Considering that each block depends on previous ones, the only type of node that can prove it it's the FullNode, which has the full blockchain. They may be a shop, a company that doesn't want to trust someone else providing them information on unspent bitcoins, etc.

  An SVP peer (Simplified Payment Verification) can't prove it on its own and it relies on a FullNode which may send invalid transactions. So the light client has to verify it.

- **And how does a Bitcoin light client verify it?**

  The light client will apply the Merkel proof, applying recursively the hashing function, starting with the received transaction to check. Then it will compare the result with the Merkle tree root stored in the block header, which is a small part that even the light client may store.

2. Pay eggs' supply chain with Ethereum smart contracts

Given the zip file of the assignment, the student needs to answer to the following tasks:

1. Implement the missing portions of the complete() function. They need to pay the transporter and, eventually, refund the penalty to the receiver;

```
function complete() public is_allowed(receiver) payable {
        uint amount = msg.value;
        require(state == State.SHIPPING, "Error, invalid initial state");
        require(amount == transporter_price, "Error, funds not match the expected amount");
        uint penalty = penalty_function.compute_penalty(samples_temperature, samples_bump,
                        temperature_threshold, bump_threshold);

        if((transporter_price < penalty) || (transporter_price - penalty) < minimal_price) {
                // The penalty overcomes the difference between initial and minimal price
                (bool success, ) = receiver.call.value(amount)("");
                require(success == true, "Error while refunding receiver");
                state = State.REFUSED;
                emit refused(penalty);
        }

        else {
                // Accept the box and pay the transporter
                uint to_pay = 0;
                to_pay = transporter_price - penalty;
                (bool success, ) = transporter.call.value(to_pay)("");
                require(success == true, "Error while paying transporter");
                (bool success2, ) = receiver.call.value(penalty)("");
                require(success2 == true, "Error while refunding penalties to the receiver");
                state = State.RECEIVED;
                emit received(to_pay);
        }
}
```

2. Evaluate the cost in gas of the functions provided by the smart contract;

*I've used Truffle and Ganache as suggested during lessons. The tests are the given ones, but since they weren't "long enough" I've simply copy-pasted the given numbers to reach a total of 96 (12*8) values.*

Results:

Contract: BoxIncomplete

ship: 53906	**ship**: 53906	**ship**: 53906
push_data: 4278387	**push_data**: 4278531	**push_data**: 4278387
complete: 236099	**complete**: 393445	**complete**: 296730
all_ok	all_not_ok	some_ok_some_not

It's possible to see that the **ship() cost** is the same no matter the failings, which is exactly what it should happen. The **push_data()** cost changes depending on the presence of penalties or not, but of course doesn't change too much. The **complete()** function on the other hand changes a lot the more penalties are introduced, because these penalties must be calculated and this implies a further gas cost.

3. Compute the fees to be paid by B to store the samples, push_data() , for a single shipment. Use a gas price suggested by the Ethereum gas station3, and the current exchange Eth-Euro.

Using https://ethgasstation.info/calculatorTxV.php for the next tables.

a. **Recall, the sensors produce a new pair of data (temperature, bump) every 5 minutes. The shipment lasts for 8 hours.**
b. **You can evaluate the total fee whether B stores on the smart contract a new pair as soon it is produced, all the pairs only once, or a combination of the twos;**

I've evaluated the total fee on the second possibility, "all the pairs only once".

In the "all_ok" test and "some_ok_some_not":

Predictions: Gas Used = 4278387; Gas Price = 31 gwei

Outcome	
% of last 200 blocks accpeting this gas price	97.7777777778
Transactions At or Above in Current Txpool	25
Mean Time to Confirm (Blocks)	102
Mean Time to Confirm (Seconds)	962
Transaction fee (ETH)	0.13263
Transaction fee (Fiat)	$32.49435

In the "not_ok" test:

Predictions: Gas Used = 4278531; Gas Price = 31 gwei

Outcome	
% of last 200 blocks accpeting this gas price	97.7777777778
Transactions At or Above in Current Txpool	25
Mean Time to Confirm (Blocks)	102
Mean Time to Confirm (Seconds)	962
Transaction fee (ETH)	0.1326345
Transaction fee (Fiat)	$32.49545

4. What are the security concerns to keep into consideration while developing a smart contract similar to this one? List the vulnerabilities that might arise. Use the provided articles, or any online resource.

a. Is this contract correct, or does it present issues? (Vulnerabilities or even programming mistakes).

I've read the given articles and researched online a bit. The problem lies within the definition of "Software" itself, because software implies bugs and vulnerabilities in the source code which are the main reasons behind the security issues in smart contracts. Furthermore, depending on what platform we're using, even Virtual Machines may be a problem. It's well known how the Ethereum Virtual Machine (EVM) is prone to several vulnerabilities: from access control issues to immutable detects.

Speaking about Ethereum, the first error that happens kind of frequently is the **integer overflow**. Let's just take a quick look at our simple smart contract developed during this final term and we can see that we're handling very big integers that will overflow. That's why I've added another check in the if condition, which is the *(transporter_price < penalty)*, because otherwise it'd reach MAX_INT and it'd never go into the *if branch*, always skipping into the *else* one. So this contract had vulnerabilities that I tried to address by handling them. This problem was used recently during the *batchOverflow* hack.

Another one is **reentrancy**: Ethereum contracts are prone to reentrancy attacks when a function is called repeatedly by a malicious external contract before the first invocation of this function was even finished. This vulnerability makes it possible to change the state of a contract in the process of this execution. The DAO hack happened because of this issue, which allows participants to donate Ether to fund contracts at their choice, so to steal all the Ether from the contract.

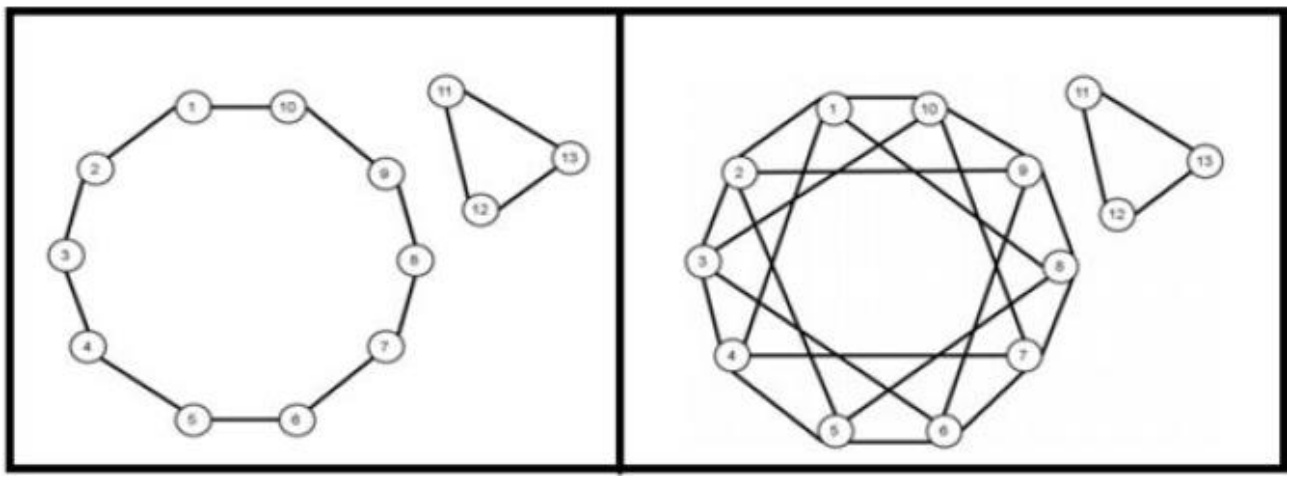
Finally, **Denial of service attacks**. A simple and used way to calling the *bid()* function constantly would prevent other users from making their bids.

Following best practices for each platform / language is a must, but errors may be unintentional too. So it's suggested to run a security audit whenever it's possible, because testing can't show the absence of bugs, even the most common ones.

About this specific contract, I couldn't see too many issues except the previously stated one, about overflow. The contract is small, direct and uses simple commands. The only command that isn't as secure as it can be, it's the *.call.value(address)* which is better to avoid when we can use *send(address)*: the problem here is that addresses must be payable to use the *send()* method, so I could only use the less safe one.

3. Complex Network Analysis

Compute the clustering coefficient, calculated as the average of the clustering coefficients of the single nodes of the graph, for the two graphs shown in Figure 1.



The clustering coefficient CC of graph G is the average of the clustering coefficients of all nodes in G.

First graph:

Clustering coefficient of the **circle** (10 nodes): **0**

Clustering coefficient **triangle** (3 nodes): **1**

Clustering coefficient **total** (3/13): **0.23**

Second graph:

Clustering coefficient **circle** (10 nodes): **0**

Clustering coefficient **triangle** (3 nodes): **1**

Clustering coefficient **total** (3/13): **0.23**

- **What do you observe? Is the value of the clustering coefficient a good measure to characterize the structures of the graphs?**

I can observe that the clustering coefficient isn't a good measure. The two graphs are different, but the clustering coefficient results in the same value for both of them, not showing in any way this possible difference. This happens because the clustering coefficient takes neighbors and connections between them only, which isn’t really representing the graph’s structure.

- **Describe a better way to characterize these structures**

During classes, we introduced the **edge density** of a graph. It's defined as the ratio of the number of edges with respect to the maximum number of possible edges.
Applying the formula, the results for the edge density are:

1st graph: 0.1666 *(periodic)*

2nd graph: 0.295 *(approximated)*

With these values, we can see already see just by the edge density itself that the second graph is denser than the first one. Furthermore, we can also see that they're actually two different graphs, so this is a good characterization.

A final note about modern literature is that we introduce the "Small World Network": generally speaking, nodes should be connected with at most 6 hops. We also introduced many ways to create short paths following the SWN idea, like Watts Strogatz or Kleinberg model (for static graphs).