EasyChair Preprint

№ 99

Blockchain technology: possible methods of transaction validation and version consensus

Brylov Alexandr, Ostrovskaya Kate and Mikhalyov Alexandr

April 28, 2018

Blockchain technology: possible methods of transaction validation and version consensus

BrylovA.O.[1] and Ostrovskaya K.Y.[2] and Mikhalyov A.I.[3]

[1]National Metallurgical Academy of Ukraine, Ukraine
S.Brylov@outlook.com

[2]PhD, National Metallurgical Academy of Ukraine, Ukraine
kuostrovskaya@gmail.com

[3]Dr.Sc. (Ing), Professor, National Metallurgical Academy of Ukraine, Ukraine
maillich2@gmail.com

Abstract. An article contains the overview of blockchain technology, major principals of its programming. Blockchain is the data structure consisted of blocks where every next block is strictly connected with previous one by including its hashcode as property. Every blockchain is also implements the decentralized storage of data. Several critical aspects are analyzed: optimal solutions for transaction validation and algorithms of consensus between nodes in blockchain decentralized network. The validation may be organized by storing the transactions data in the separate nodes and by including the "bookkeeper" class designed to review the transactions and validate it. Alternative approach is to make the transaction"self-validating", so it is always consists entire data required for validation. There are several ways for reaching the consensus across network about which version of blockchain is consider as correct one. Simple rule of "longest and late prevails" is suitable for small, non-scalable projects, while rule of "50%+1" needs to be implemented for larger ones. Non of those solves the problem of wasting of resources, so this aspect remains a subject for further research and testing.

Keywords:blockchain, validation, algorithms.

The word "blockchain" attracts significant attention of programmers, investors and, overall, a wider society, people who watch for the innovations in IT industry. The major part of this interest is resulted by fast monetizing of cryptocurrencies, unprecedented appreciation of bitcoin, and, soon later, of the other coins, though is not limited to this only. The consensus among financials already exists that cryptocurrencies are going to take a definite place in the world economy, while experts in programming and engineering are confident that the blockchain is breakthrough technology that can be applied in almost every important area of life and society [1].

Let me briefly describe the core of blockchain. In general, the blockchain is nothing more than data structure – the list or linked list (in C# terms), that implements the set of definite rules. The block of blockchain is an entity, that stores the data and several

other attributes. The data stored inside the block are usually called “transaction” even if this specific project does not connect to finance and cryptocurrency. So, in this article I will use terms transaction, block, blockchain.
In the general case the block consists of transactions (one, two or many – it does not matter and implemented according to the specific task), timestamp and, the most important, the previous block. Obviously, such concept leads to progressive size growth of each next block and, thereafter, its size will exceed soon the limitations of network data exchange. That is why one of the key feature of any blockchain implementation is hashing of data.
Any data, regardless of data type (integers, symbols, strings, objects) can be casted to common format, usually – to string. Moreover, any string of arbitrary length, using crypto-algorithms, can be casted to array of symbols of fixed length (bit string). Such algorithms usually called “hashing”: applying hashing to the input string results an output string of fixed length. Changing of even single character in the input string always results the completely different output string.
Thanks to above method, there is an opportunity to avoid necessity to include an entire previous block into the next one. Instead, hash-code of previous block is included only. This should be noted here, that hash does not allow decryption of input data and it is not designed for. The most important in this concept is the fact, that attempt to change any piece of data of previous block results in the change of its hash, hence the incompatibility with the next block.

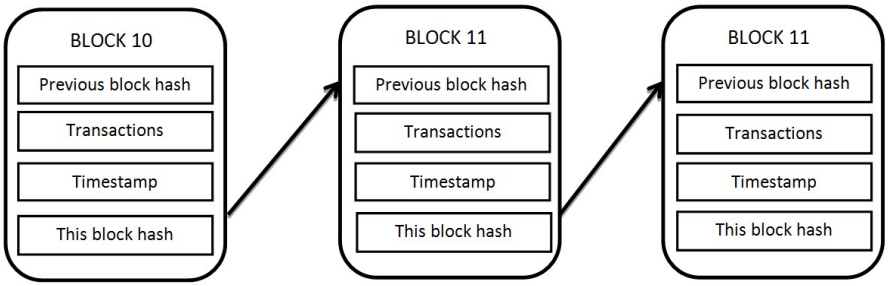

Fig. 1. - Block structure and blockchain build

Hash of current block is not included into the block but is a resulting string of its entire content. It is written into the next block. Therefore, any changes made in block 10 lead to the situation when its output hash does not equal any more to the previous hash of block 11. Since the verification of hashes equality is simple and does not require large computing resources, the integrity of the entire blockchain is simple and can be monitored instantly. Thanks to that, blockchain has a high level of security[7].
In general case the programming code of the block creation (constructor) is implemented as following:

```
public Block(inti, Transaction transactionData, string
prev = "")

        {
```

```
            index = i;
            Data = transactionData;
            timestamp = DateTime.Now;
PreviousHash = prev;
            Hash = CalculateHash();
        }
```

where the most important method – CalculateHash() – may be defined in the way:

```
public string CalculateHash()
        {
StringBuildersb = new StringBuilder();
            string toEncrypt = index.ToString() + Pre-
viousHash + timestamp.ToString() + Data.ToString() +
nonce;
            using (SHA256 hash = SHA256Managed.Create())
            {
                Encoding enc = Encoding.UTF8;
                Byte[] result =
hash.ComputeHash(enc.GetBytes(toEncrypt));
foreach(Byte b in result)
                {
sb.Append(b.ToString("x2"));
                }
            }
            return sb.ToString();
        }
```

The second most important principal of the majority of blockchain projects is decentralization of data storage. Naturally for human, when one needs to protect something valuable, he tries to hide it the most secured place and lock it behind as many lockers as possible. But, in case intruder succeeds to break the protection, there is nothing

more to prevent him of destruction or changing the data. Even more, in some circumstances the fact of storage penetration is not always obvious. The blockchain projects implement the opposite approach. The copies of data are stored at multiple separate nodes. Data changes in the one of the node do not automatically lead to data changes in the others. Vice versa, as soon as the discrepancy is identified, the node, that has been changed (or hacked), restores the original data by synchronizing with the peers. The probability of simultaneous and synchronized intervention into the multiple nodes is significantly lower and, is becoming even lower as the number of network participants is growing[2].

There are several other principals used in the majority of blockchain projects: proof-of-work – applying a deterministic requirements to the hash of block (for example, it must start with several zeros), proof-of-stake and other methods of confirming of the data transaction contains, smart-contracts, but most of such principals are not necessary and may have different implementation according to specific project needs.

In order to describe the most basic issues blockchain-programmer has to solve, it is worth to give a brief overview of the structure of decentralized network which will use this project: who are participants, what are their roles and functions.

In common practice there are three types of participants: nodes, miners, users.

Nodes store data and permanently broadcast them across the network. Users make transactions. Miners create blocks. The last type called “miners” because the majority of blockchain projects implement proof-of-work principal, which designed to prevent uncontrolled block emission. The difficult mathematic problem requiring substantial computer power must be solved in order to create new block. But in general sense, the miners are nothing more than nodes, that serve the blockchain [7].

Users make transactions (not necessarily in financial meaning. It might be contracts, file creation, voting, so on). Nodes receive transactions and broadcast them. Miners collect transactions and create blocks that include transactions and pass them to the nearest nodes. The node gets new block, verifies it, attaches it to blockchain and broadcasts updated blockchain to peer-nodes.

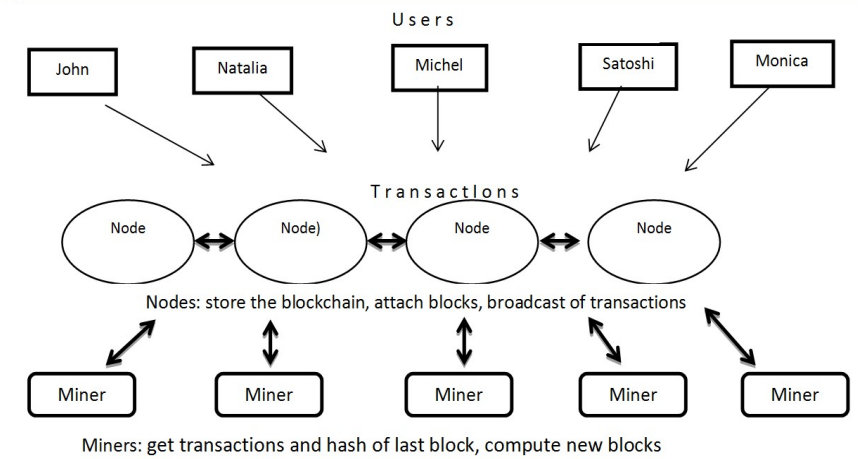

Fig. 2. - General map of interaction of network participants

Obviously, the idea does not look complicated as well as its programming implementation. But the software developer who works on such project, faces several issues right at the beginning. These issues might have a variety of possible solutions and, the purpose of this article is to review such solutions. Though, as I mentioned above the blockchain technology is not limited to cryptocurrencies only, but has a variety of applications, in the examples below I will use cryptomoney examples and terminology because it allows to describe the issues easier and more clearly. Therefore my examples are based on the possible implementation of blockchain for finance transactions purpose.
The first problem is related to methods and algorithms of validation of transactions. For example, user John wants to send to user Natalia 10 cryptocoins. Using some user interface he prepares and completes the transaction. The nearest functioning node receives this transaction. Evidently, such node needs to validate this transaction before broadcasting. In the opposite case, future blocks might include incorrect transactions so the miners computing power is wasted.
The very basic transaction includes five properties: sender address, receiver address, amount and timestamp. In order to make a validation easier, we also include the property TransactionHash, which represents the hash of other properties.

```
public class Transaction

{

        public string Sender { get; set; }

        public string Recipient { get; set; }

        public decimal Amount { get; set; }

        public DateTime Timestamp { get; set; }

        public string TransacationHash { get; set; }

}
```

Thus the validation of transaction consists of answering three questions: does receiver exist, does sender have sufficient funds, and, is current transaction unique (does not exist among completed transactions).
In order to work out the algorithm, we need to decide first where and how users data are stored (accounts usually represent the string of bit of fixed length), and, completed transactions data. While developing a blockchain project, I tested two possible approaches:

1. Some nodes play a role of “bookkeeper”: contain the list of registered accounts and, separately, list of completed and verified transactions. Obviously, such nodes are also instantly synchronized between each other.

2. User’s accounts included in the blockchain. Every block has an entire list of registered accounts.

The benefits of first variant are the following: blockchain is less loaded with the data, every block is smaller, and, actually, the transaction validation is simple, programming code is shorter. But there are some disadvantages as well: additional objects are created, and we need to work on synchronization, security of such objects and, potentially, the network becomes less decentralized.
The second variant is more elegant but demands more complicated validation methods and leads to increase of block size.
So the validation of transaction starts with verification of existence of receiver. In case registered accounts are stored in separate list, such list must be iterated while match is not found. If user's data stored in the blockchain, the node must iterate an entire blockchain while match is not found.

```
bool IsValidAddresses(Transaction someTransaction,
List<Account>ListOfAllAccounts)

{

            if (someTransaction.Sender == some-
Transaction.Recipient) return false;

            bool isValidSender = false;

            bool isValidRecipient = false;

foreach(Account account in ListOfAllAccounts)

            {

                if (someTransaction.Sender == ac-
count.Address) isValidSender = true;

                if (someTransaction.Recipient == ac-
count.Address) isValidRecipient = true;

            }

            if (isValidRecipient&&isValidSender) return
true;

            else return false;

 }
```

The verification of funds sufficiency is more interesting task. In traditional banking service, banks always store the balance of each account: amount of funds available for spending. So in order to complete a fraudulent transaction (spend an amount exceeding the available funds), one number only must be changed. Such a threat contradicts to principals of blockchain. In order to make validation more reliable, we might get amounts of every incoming and outcoming transactions related to specific user. The positive difference between sum of first and latter is the criterion of correct transac-

tion. Such approach requires the iteration of the entire array of past transactions, whether they are stored in the separate list or included in the blocks.

```
public bool IsValidTransaction(Transaction some-
Transaction, Hashtable transactions,
List<Account>ListOfAllAccounts)

{

            if (!IsValidAddresses(someTransaction, ListO-
fAllAccounts)) return false;

            decimal amountRecieved = 0;

foreach(Transaction instance in TransactionsAsReci-
pent(someTransaction, transactions))

            {

amountRecieved += instance.Amount;

            }

            decimal amountSent = 0;

foreach (Transaction instance in TransactionsAsSend-
er(someTransaction, transactions))

            {

amountSent += instance.Amount;

            }

            return ((amountRecieved - amountSent - some-
Transaction.Amount) >= 0);

  }
```

Bitcoin project implements another interesting and elegant method of validation. It implements so-called self-sufficient transaction architecture, the transaction that contains all data required for funds verification. In such concept the transaction might have a link to several incoming and outcoming transactions. The amount of outcoming must not exceed the amount of incoming. When user John has 100 coins and sends to Natalia 10 coins, the interface automatically generates an extra transaction: John sends the remaining 90 coins to himself. When John decides to make next transaction, the 90-coins-transaction from previous one appears as incoming transaction [6].

The only drawback of such approach is the complexity of transaction, as well as the fact that it requires additional applications on user side, which will automatically

create transactions to self. But overall, this is an excellent example of object oriented approach when object itself is self-sufficient.
Another problem, that I want to review in this article, are methods and algorithms of consensus between nodes regarding to which variant of blockchain to consider as correct one [5]. Any network communication works with some delay of data transfer. Data arrive to every participant assynchronically, not at the same moment of time. Closer node might get updated data sooner than the remote node. Let's imagine, that two miners, connected to different nodes, started to work on new block simultaneously. By some reason, first miner (let's call him "Miner A") has finished the job and passed the newly created block to his nearest node ("Node A"), and, started to compute the next block. The node, who has got the block, attached it to blockchain and started to broadcast updated blockchain across network. A few milliseconds later, second miner ("Miner B") finished his job as well and passed it to his nearest node ("Node B"). This has happened just one millisecond, but still before Node B receives an updated version of blockchain from Node A. So, Node B also attaches new block (created by Miner B) to his, old version of blockchain and started broadcasting it as well. One of rules of blockchain is "The longer chain prevails", that means that nodes must consider the longest blockchain version (the blockchain containing the greatest number of blocks) as correct. But in our case the versions at Node A and B are still having same length though the last block is different. Such example is called "blockchain conflict". When (or if) Miner A creates new block, Node A attaches it and starts broadcasting before Node B gets next block from Miner B, Node B has to accept the most recent version of blockchain broadcasted by Node A, as correct. Now on, the blockchain is same at both nodes and conflict has resolved. But Miner B keeps working on his second block though it will not be valid anymore because its previous hash is not equal to hash of most recent block of blockchain. So resources of Miner B has been wasted, wasted not only for last block, but for previous one as well.

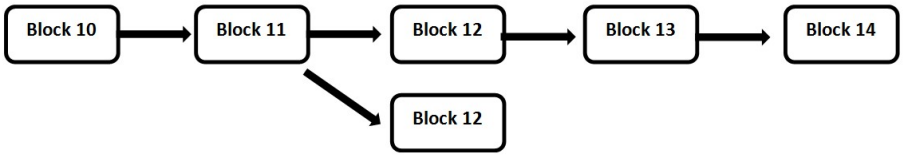

Fig. 3. - Short blockchain conflict

This problem has two sides: first – the definition of optimal algorithm of consensus, second – optimizing of usage of computing resources[4].
If the network is small, number of participants and transaction frequency is expected to remain limited, then the very simple algorithm can be implemented. The block creation shall be engineered in a such way so the average block creation time is substantially longer than the time required for every node of network to receive an update of latest blockchain version. Then we can define the frequency the nodes to synchronize: each node gets blockchain version from every other node. The node can then check the length of each version and save the longest one.

An extra criterion applied in case of existence of several versions with the same length: for example, timestamp of last block. But, this approach does not satisfy the requirements of scalability and it can't be implemented if data transfer time between remote nodes exceeds new block creation time.
The full synchronizing between nodes might take much longer time the speed of blockchain updating in separate nodes in the larger projects, that designed to serve the nodes located very far from each other. In such case, we might face a scenario when alternative versions of blockchain might appear and keep living for long time inside the remote segments of the network. Such event called "soft-fork". As shown in the above example, that means the large amount of blocks have to be wasted when finally the consensus is achieved. Therefore, many transactions that have been included already in that thrown away blocks, become unconfirmed again and must be re-worked into new blocks. The partial solution (or at least some optimizing) is to include additional property into the block structure: number of confirmation, i.e. the number of nodes that have accepted this block as valid and have added it to blockchain. This reduces the accumulation of conflicts: consensus between the nodes to be found much earlier than the length of some node's version becomes bigger. Finally, it implements the "50%+1" principal: if large soft-fork has happened the network finally accepts the blockchain variant that collects more than 50% of confirmations [6].

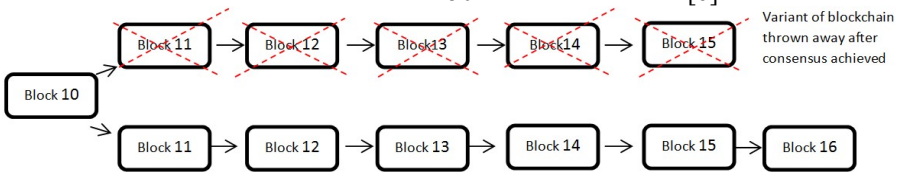

Fig. 4. - Extended conflict – soft-fork

But the sad issue of multiple wasted blocks still remains. I can see two ideas for future research and testing. The first is, let say, "planned soft-fork". Transactions are to be pre-definitely distributed between miners for further computing. Miners create blocks that are attached by the nodes to its blockchain versions. Soft-forks are allowed for some determined length, for example, 10 blocks. When several versions have achieved this length, the computing of new blocks has to be stopped until one special joining block is created. Such special block must include hash of hashes of all blockchain versions and pointers to the latest blocks of both.

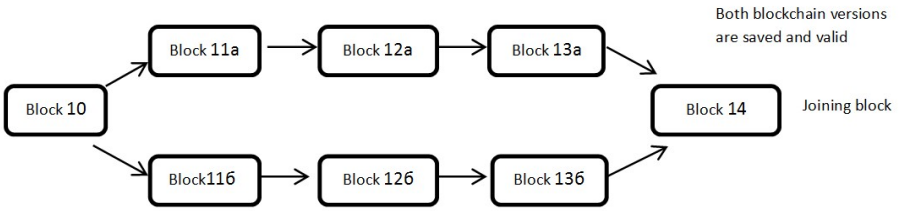

Fig. 5. - Planned soft-fork

The second idea is to build the proof-of-work algorithm in a such way so the mathematical problem that has to be solved to create the new block can be divided into parts and those parts to be distributed between miners. It can be possible to find an algorithm, that allows to pre-compute the block independently on the transactions to be recorded in. This makes possible to use the computing job done once to be applied to any future block.

Both ideas are not yet implemented and require future analysis and discussion as well as deep testing to determine the complexity, resources required and fault tolerance. At high extent, this definition is valid for the entire blockchain technology, because, despite hundreds and thousands of projects already launched, technology is still on the stage of its birth. Evidently, the potential is great, the sphere of application is very wide [3]. Therefore, I believe that development and further research of blockchain is an excellent area of study and work for at least one entire generation of software engineers.

References

1. Tapscott Don, Tapscott Alex. Blockchain revolution. New York, 2016.
2. Antonopoulos Andreas M. Mastering Bitcoin. Programming the open blockchain. New York, 2017.
3. Vigna Paul, Casey Michael J. The Age of Cryptocurrency: How Bitcoin and the Blockchain Are Challenging the Global Economic Order. New York, 2016.
4. Wattenhover Roger. The Science of the Blockchain. New York, 2016.
5. Pease Marshall, Shostak Robert. The Byzantine Generals Problem. ACM Transactions on Programming Languages and Systems. 1982.
6. Satoshi Nakamoto. Bitcoin:A Peer-to-Peer Electronic Cash System. Bitcoin whitepaper at https://bitcoin.org/bitcoin.pdf.
7. Dorier Nicolas. Blockchain Programming in C#. https:// programmingblockchain. gitbooks.io/.