A primer on blockchain design

Dmitrii Zhelezov

November 23, 2018

1 Introduction

A typical blockchain lifecycle consists of

1. Peer discovery and transaction broadcasting (usually using a gossip protocol)
2. Transaction block generation and broadcasting
3. Block verification and local state updates by network nodes
4. Network consensus on a new global state

In this note we will concentrate on the last three steps and discuss main challenges a modern blockchain implementation has to tackle. In particular, we'll see how the much hyped but often poorly understood[1] concepts PoW (Proof of Work), PoS (Proof of Stake), DAG (Directed Acyclic Graph) fit into a general blockchain design and why building a robust blockchain network is hard.

While mostly informal, I tried to find a balance between readability and detalization. The goal was to make an exposition which is easier to read than an typical academic paper yet more detailed than a typical (video)blog. It's inevitable that many important points are missed or poorly explained, so I recommend to use complementary materials, e.g. an excellent survey [13] with graphical explanations of the inner workings of distributed systems.

2 TL;DR

1. Section 4. Consensus is needed since each participant has it own slightly different view of the world.
2. Section 5. Finality is needed to prevent double-spends.
3. Section 6. If you can flood the network with many clones (Sybils) supporting you, you can revert the consensus. Finality is incompatible with Sybil attacks, thus Sybil attacks should be hard.
4. Section 7. PoW is a Sybil resistance scheme (not a consensus protocol): one has to spend a lot of resources to propose a block, so a Sybil attack is expensive.
5. Section 7. Nakamoto consensus is a static rule: take the chain with the highest difficulty. No network communication is needed.
6. Section 8. PoS is another Sybil resistance scheme: block producers are pre-defined, one has to spend the internal currency in order to have a chance to become a block producer. Thus, producing many blocks for a Sybil attack is expensive.
7. Section 9. By using PoW or PoS one can make the network is synchronous if the average block proposal time is larger than the network latency.
8. Section 9. High block proposal rates are only possible in asynchronous networks with many forks. DAG is a data structure which allows a block to have many parents, efficiently keeping track of many intertwined forks. In general, if two blocks are conflicting, the one included in more (weighted) forks becomes canonical.

[1] An excellent Twitter-based #microlecture [20] by Emin Gün Sirer highlighted the lack of a common reference and served as an inspiration.

9. Section 9. Building a Sybil-resistant decentralized asynchronous system with fast transaction confirmations (time to finality) is hard.

3 Local views vs Global state

A blockchain is a distributed system whose state $S(t)$ evolves over time t, which we will treat as time in seconds since the "genesis" event. A classical example of a state $S(t)$ is a ledger with account balances[2] at time t, but it really can be anything (e.g. in Ethereum, it's the state of EVM, so in particular it contains all the states of all smart contracts in the network). However, it's tricky to define a global state S for a distributed system and here is why.

Each node u participating in the network has its own *view* $S_u(t)$ of the state at time t, so there are actually as many versions of the state as the number of nodes in the network. In order to define the global state $S(t)$ of the blockchain, we need to have an efficient consensus protocol which would allow all the participating nodes to agree on the same state and update their local views accordingly. In other words, the global state $S(t)$ is a version of the view at time t everyone agrees upon.

Even worse, in *inclusive* systems (i.e. anyone is free to join the network) some nodes may be malicious and try to interfere with the network. In this case we still want that the virtuous nodes, faithfully following the protocol, reach a consensus about $S(t)$. The exact network security requirements may vary and real world deployments should take into account economic incentives of the participants. See [8] for further discussion about realistic frameworks.

4 State as a chain of blocks

The network nodes are going to reach consensus not only about the current global state, but also about the history $S(t)$ for each time $t = [0, \dots, T]$ up to the present time T.

A natural approach is to represent a state $S(t)$ as a sequence of blocks $B_1, \dots, B_{t-1}$ where the block B_i contains all transactions included in $S(i+1)$, but not in $S(i)$. One can encode this as

$$S(i+1) = S(i) \oplus B_{i-1},$$

where $S(i) \oplus B_{i-1}$ means "apply the transactions contained in B_{i-1} to state $S(i)$". Then

$$S(t) = B_0 \oplus B_1 \oplus \dots \oplus B_{t-1}.$$

In particular, in order to reach consensus on $S(t)$, the network agrees on some sequence of transaction blocks of length $t-1$

$$(B_1, \dots, B_{t-1}).$$

Algorithm 1 illustrates such an approach. Note that the most recent t for which the local state $S(t)$ is known, as well as the ordering of the blocks, is completely determined by the routine chooseBlockSequence(), which should be considered as a block-box for now. A concrete implementation is a crucial part of any distributed ledger protocol which to a great extent determines the protocol security and robustness.

In Section 7 we will consider an particularly elegant and efficient implementation of chooseBlockSequence(), suggested by Satoshi Nakamoto for Bitcoin.

Now assume we live in an ideal world where

A0 The routine chooseBlockSequence() always returns a valid sequence of blocks, i.e. applyBlock() at line 8 never fails and returns a valid state.

A1 All nodes have the same version of $\mathcal{B}$ at the time a new block is produced.

A2 All nodes faithfully follow Algorithm 1.

Then all nodes in the network have the same local view $S(t)$ for times t up to the time the last block had arrived, since chooseBlockSequence() is executed by each node with the same input.

[2]Instead, a ledger may keep track of who owns UXTOs (unspent transaction outputs). This model is used by e.g. Bitcoin. See [10] for further explanation and discussion.

Algorithm 1 Update local view.

```
B_0 ← genesis
$\mathcal{B} \leftarrow \{B_0\}$                                ▷ Collection of all blocks
$S[0] \leftarrow \emptyset$                                      ▷ Starting state
while true do
    if $B \leftarrow$ newBlock() then
        addBlock($B$)                                            ▷ New block received
        $bseq \leftarrow$ chooseBlockSequence($\mathcal{B}$)
        loop $i = 1$ to $length(bseq)$
            $S[i] =$ applyBlock($S[i-1], bseq[i]$)
        end loop
    end if
end while
```

5 Double spend attacks

It turns out that even in an ideal world where the assumptions (A0), (A1), (A2) hold, the network is susceptible to the following double-spend attack.

Assume for concreteness that chooseBlockSequence($\mathcal{B}$) is implemented as follows: it outputs the longest valid sequence of blocks in $\mathcal{B}$[3].

Let Alice have \$1 on her account which she transfers to Bob in return for goodies. Alice's transaction T_B is included in a block B_1. Bob observes B_1 (and hence T_B) in the global state by querying one of the nodes and sends the goodies. Next, Alice issues an another transaction T'_B which sends \$1 to herself and submits to the network. She keeps submitting T'_B to the network so that the fresh blocks include T'_B[4]. Note that if Alice has \$1 on her account, T'_B remains a valid transaction no matter how many times it had already been applied.

Sooner or later, each local copy of $\mathcal{B}$ will contain mostly blocks containing T'_B. At this point, chooseBlockSequence($\mathcal{B}$), taking the longest valid sequence of blocks in $\mathcal{B}$, is going to include some blocks with T'_B in the output sequence. But since B_1 conflicts with T'_B, B_1 is no longer included in a local view as long as at least one of the blocks with T'_B is picked up by chooseBlockSequence($\mathcal{B}$). Thus, B_1 will eventually be excluded from all the local views and the global state, as if T_B had never been submitted: Alice has both \$1 on her account and the goodies.

There are two general ways[5] to tackle the double spend attack described above.

(EL) **Elitarian:** Regulate the way new blocks are produced so that Alice's conflicting transactions T'_B can not be included in new blocks after the initial transaction T_B is accepted as valid by the block producer.

(EG) **Egalitarian:** Any block can be produced, but chooseBlockSequence() guarantees the following.

(F) If at time T a block B is included in a local view $S_v(i)$ for some $i \leq T - t_0$, then it's *final*, i.e. it will be always included in $S_v(i)$ in the future. It follows that once B had been accepted in the global state since t_0 seconds ago, it will remain in the global state forever.

For example, if $t_0 = 5$, then in the double-spend attack above Bob can simply wait 5 seconds (till T_B is "confirmed" in blockchain parlance), and after that it's guaranteed that T_B will stay in the global state forever. The confirmation time t_0 is arguably the most important metric of a general-purpose cryptocurrency built on top of blockchain.

Note the (EL) is stronger than (EG). Indeed, if (EL) holds, then no two blocks contain conflicting transactions. The implementation of chooseBlockSequence($\mathcal{B}$) is trivial: simply output all the blocks in $\mathcal{B}$ in the order they have been produced.

[3] This implementation is similar to the one considered in Section 7 for the sake of concreteness. The outlined double-spend attack, however, is generic.

[4] We assume that all transactions submitted to the network are eventually included in some block.

[5] It's ideal world scenarios, in practice one should always replace "always" with "likely", and "never" or "can't" with "unlikely", with the exact meaning depending on the concrete setup.

An extreme scenario for the case (EL) is dictatorship: only a single node is allowed to produce blocks (e.g. only blocks cryptographically signed by a specific block producer are deemed valid), which essentially reduces the network to a distributed database with a single master and a bunch of read-only slaves. Indeed, centralized systems, e.g. VISA, are much more effective than present-generation blockchains[6] in terms of transaction speed and finality times. The problem is that the block producer should be always online and *trusted*. A malicious block producer can censor out valid transactions[7] or even go rogue and produce conflicting blocks. A more advanced and less centralized approach is to form committees who vote for block candidates, the winning block candidate becomes official[8]. Such an approach gains popularity in combination with the Proof-of-Stake block proposal mechanism, discussed in Section 8. For a thorough discussion of centralisation vs scalability and an overview of existing projects in the space, see [13]. For a philosophical and economical take on the problem of censorship see [9].

6 Sybil attacks

Systems implementing (F) may be completely inclusive, in a sense that any participant can leave or join the network at any time and produce blocks. It opens up a much larger set of attack vectors than in the elitarian approach. In particular, as we will shortly see, there must be a mechanism preventing an adversary to easily produce a large number of blocks and succeed in a *Sybil attack*.

Recall Alice from Section 5 who submits a transaction T_B with money to Bob, and asks goodies from Bob in return. Bob, now well aware of double spend attacks, is waiting until T_B is deemed final. However, it turns out that if Alice can produce as many blocks as she wants, Bob can never be sure that T_B is final and his hard-earned money is safe.

Assume T_B is in block B_1 as before. First, Alice makes another block B'_1 by removing T_B from B_1 and broadcasts to the network. Next, each time a new block B_i is produced, Alice forges her own block B'_i by including only transactions compatible with B'_1 representing her "alternative truth".

Now for all times t there are two competing sequences of blocks:

$$(\dots, B_1, B_2, \dots, B_t) \tag{CANONICAL}$$

which contains T_B, and

$$(\dots, B'_1, B'_2, \dots, B'_t), \tag{ALICE}$$

prepared by Alice. In (ALICE) Alice has goodies for free since there are no transaction to Bob's account.

The problem now is that chooseBlockSequence() is a deterministic function, and there is no *a priori* way to guarantee that (CANONICAL) is always preferred over (ALICE), even if B_1 was produced well before Alice's blocks.

In fact, (CANONICAL) and (ALICE) are two equally valid alternative versions (called *forks*) of the global state and the conflict cannot be resolved without any additional criteria distinguishing the "fair" blocks in (CANONICAL) from (ALICE).

Summing up, if anyone can easily produce a valid block, nothing prevents Alice from building a plausible version of an alternative global state and break finality. Thus, even in open inclusive networks there should be some *block proposal* mechanism that hinders mass production of legit blocks by a single participant.

7 Nakomoto consensus and PoW

Skipping the glorious history[9] of Proof-of-Work (aka PoW) with necessary kudos to Cynthia Dwork, Moni Naor (who invented Proof-of-Work) and Satoshi Nakomoto (who invented blockchain on top of it), let's get straight to why PoW is such a good solution against double spend and Sybil attacks. For a detailed description of the Bitcoin protocol, see [17].

Satoshi's design relies on the following features.

[6] VISA can allegedly handle 24000 transactions per second (tps), in comparison to 15-25tps of the Ethereum network. See e.g. [18] and references therein.

[7] Think of VISA blocking transactions from your account.

[8] One of the notable examples is the EOS blockchain with 21 designated block producers elected in advance by EOS token holders.

[9] See e.g. [14]

(T) Each block contains a hash of it's parent block.

(D) Each block B has *difficulty* $d(B)$, which is essentially

$$d(B) = 2^{\text{number of leading zeros of } hash(B)}$$

Since *hash* is pseudorandom, it's computationally hard to produce blocks with high difficulty: brute force is the only way.

The property (T) is very useful: it naturally induces a tree-like structure to the collection of blocks $\mathcal{B}$, with each block B having a single parent block $p(B)$. Thus, implementing $\textsc{chooseBlockSequence}(\mathcal{B})$ becomes more pleasant: it suffices to pick a leaf block and reconstruct the sequence by traversing the tree up the genesis block B_0.

The property (D) provides a simple yet powerful way to choose forks: the one with higher total difficulty wins. Both ideas are illustrated by Algorithm 2 and Algorithm 3.

Algorithm 2 Local tree update

```
 1: $B_0.height \leftarrow 0$
 2: $B_0.chainDifficulty \leftarrow 0$
 3: $B_0.difficulty \leftarrow 0$
 4: procedure $\textsc{addBlock}(B)$
 5:     $pHash \leftarrow B.pHash$
 6:     if $\mathcal{B}.contains(pHash)$ then
 7:         $parent \leftarrow \mathcal{B}.get(pHash)$
 8:     else                          ▷ Parent block is missing, request from the peers
 9:         $parent \leftarrow \textsc{download}(pHash)$
10:         $\textsc{addBlock}(parent)$
11:     end if
12:     $B.parent \leftarrow parent$
13:     $B.chainDifficulty \leftarrow parent.chainDifficulty + B.difficulty$
14:     $B.height \leftarrow parent.height + 1$          ▷ Height is the distance to the root $B_0$
15:     $\mathcal{B} \leftarrow \mathcal{B} \cup B$
16: end procedure
```

Algorithm 3 Nakamoto consensus: the fork with the largest difficulty wins

```
1: function $\textsc{chooseBlockSequence}(\mathcal{B})$
2:     $B \leftarrow \textsc{argmax}(B.chainDifficulty$ for $B$ in $\mathcal{B})$
3:     return $\textsc{traverse}(B)$          ▷ Return the sequence of blocks from $B_0$ to $B$
4: end function
```

It's worth noting that the *consensus* protocol itself is simply an *a priori* agreement between virtuous nodes to follow Algorithm 3. In principle, the block difficulty can be substituted with any function which gives a total block ordering, e.g. by comparing block timestamps. It would work as far as participants are fair albeit very vulnerable to rogue participants, like Alice in Section 6. Proof-Of-Work is a mechanism that makes the Nakamoto consensus resistant to Sybil attacks by making producing legit blocks hard. Strictly speaking, it has nothing to do with the consensus itself.

So why and how exactly does Nakamoto consensus imply finality? In fact, it does not provide 100% finality guarantee, but rather says that an adversary should control at least 50% of the total network computational power in order to produce a long legit fork[10].

In Bitcoin, Ethereum and other PoW-based blockchains the blocks are produced by *miners*, who produce (aka *mine*) new blocks on top of the existing ones. Each block contains a special transaction which sends a reward to the miner address, together with all the fees from the transactions sealed in the block. Only the blocks forming the global state represent the "truth", so in order to get the rewards, a miner should produce a valid block with high difficulty.

[10]Or, to be precise, that the probability of a successful fork of length k for an adversary with less than 50% of computational power drops exponentially in k.

Thus, miners in *general* mine their blocks on top of the longest chain, so an adversary has to compete with the rest of the network in order to build an equally long fork[11]. In particular, if an adversary has less than 50% of the total computing power, the "collaborative" branch always grows faster (on average) than any alternative branch built by a sole miner. In particular, in order to build a legitimate fork (ALICE), Alice has to spend at least as much resources as the whole network has spent on the blocks in (CANONICAL) mined on top of B_1. A rule-of-thumb for Bitcoin is that it is relatively safe to consider a block final if 60 blocks were built on top of it[1].

8 Proof of Stake

Proof-of-Work is an exceptionally stable block proposal scheme, and it's still the *only* battle-tested mechanism known to work well. The crucial problem with PoW is the tremendous amount of resources being spent on forging blocks with high difficulty [2]. The main goal of Proof-of-Stake is to replicate the mechanics of PoW (based on the fact that computing power is expensive) with economic incentives – based purely on the underlying cryptoasset.

Remember that PoW systems give only conditional finality of a block B by claiming that

(1) Either B is part of the canonical chain

(2) Or someone has spent an overwhelming amount of resources[12] in order to outperform the difficulty of the canonical chain built on top of B.

PoS systems replace it with "economic finality" by promising that [3]

(1) Either B is part of the canonical chain

(2) Or large number of people have deliberately burnt very large amount of money.

In a nutshell, PoS works as follows. A user submits a "stake" using a special transaction, which locks up some funds in the deposit. In return, the user gets a chance to produce a block (the exact circumstances on who and when produces blocks may vary greatly, see [3]), she typically gets a reward, similarly to the mining reward in the PoW scheme. The higher the stake, the higher are the chances to become a block producer for the next round and get a mining reward. The fork choice rule (i.e. the implementation of $\textsc{chooseBlockSequence}()$) is also similar to Nakamoto consensus – take the branch with the largest stake associated to it[13].

However, there's an important ontological differences between the PoW and PoS approaches. PoW is an egalitarian approach since it relies on a universal *external* resource (computational power) in order to gauge block proposals, finality is achieved as a by-product of rational behavior. PoS, however, relies on an *internal* resource (underlying cryptoasset), explicitly narrowing the set of block producers for each block[14].

PoS may seem like a perfect setup for preventing Sybil attacks: the probability that Alice mines a new block is roughly proportional to the stake she has, similar to the PoW case where the probability is proportional to the share of the total network computational resources.

However, since producing PoS blocks is *cheap*, nothing prevents Alice from producing two blocks simultaneously on top of (CANONICAL) and (ALICE). Even worse, all subsequent block producers, having no prior knowledge whether (CANONICAL) or (ALICE) is going to become the canonical chain, will mine their blocks on top of both chains in order to guarantee that their block is always included and secure the mining reward. It then follows that Alice can oust (CANONICAL) having substantially less than half of the total network resources (i.e. total stakes).

This problem, known as *nothing-at-stake*, is rather serious and considerably complicates any PoS design. The general line of thought is to burn the producer's deposit if it is possible to give a cryptographic evidence that the block producer was cheating, e.g. by providing two conflicting

[11]See [15] and [8] for further discussion related to the skewed mining incentives known as "selfish mining" and "uncle" block rewards. The GHOST protocol [21], partially implemented in Ethereum, uses a more secure modification of the "longest chain wins" rule.

[12]Sufficient to power Iceland for a year.

[13]It might be tweaked due to technical reasons, like in Casper FFG [23]. The premise remains – the canonical branch concentrates the deposit value.

[14]Note that we use the terms elitarian (EL) and egalitarian (EG) strictly in the sense of Section 5. PoW systems (like Bitcoin) are often elitarian in the usual economical sense as they require large investments in hardware in order to participate in mining.

blocks signed by the same block producer. There are extra complications coming from the fact that the block producers come and go, so it's possible to punish a malicious block producers only as long as the staking deposit is locked. The implementation details are beyond the scope of the note, so we refer the interested reader to [3], where different flavours of PoS are discussed.

9 DAGs and partial order

The systems considered so far were *synchronous*, which means that most of the miners have the same collection of blocks $\mathcal{B}$ at the time a new block is produced. Both PoW (by adjusting the threshold for block difficulty) and PoS (by explicitly assigning time windows for block proposals) schemes are designed in such a way that the average time till a new valid block is proposed exceeds the network latency with some room to spare.

Still, if it happens that two valid blocks are proposed within a short period of time, it's inevitable that at least one of them is not included in the canonical chain (or *orphaned* in blockchain parlance) since they both have the same parent block. Thus, the transaction throughput in synchronous single-chain systems is inevitably capped by the number of transactions in the block divided by the network latency. This is a serious challenge, since increasing the maximal block size increases[15] latency and may compromise the protocol security (see e.g. [21]).

In order to increase the throughput, once has to weaken the synchronicity condition (A1) of Section 4: honest nodes may no longer have the same local data $\mathcal{B}$ at the time a new block is produced. On the other hand, it is natural to expect the nodes likely share a large portion of information, and *eventually* any block or transaction will be received by each node. In what follows we will still assume for the sake of consistency that transactions are batched in blocks as before, though in many cases one can do single transactions in the same fashion, e.g. [19].

Recall that in the PoW setup a rational miner, in order to maximize the chances that her block is included in the canonical chain, would mine her block on top of the longest chain. In other words, by choosing where to mine a new block, a miner places a bet (equivalent to the computational resources being spent) that the parent block is the last in the global state and that she will be the first to propose a new valid block on top of it.

In fact, it is a winner-takes-it-all situation:

1. only a single block is accepted;
2. the more computational power one has the higher are chances to win the bet;
3. participants with low computational power are virtually out of the game.

The idea is to allow the participants to place multiple bets, so that a new block can endorse multiple parents at the same time. In order to accommodate this, it is convenient to use another *data structure* for the collection of blocks $\mathcal{B}$. Up until now, $\mathcal{B}$ had a tree structure: each block, except for the genesis block B_0, had a single parent. Now we simply allow multiple parents, so $\mathcal{B}$ becomes a directied acyclyc graph (DAG) (see e.g. [4] for a formal definition and general background).

The philosophy behind the DAG approach is that having multiple parents, each block is included in a large amount of small interconnected forks. Such a complex entangled system, continuously updated by independent rational actors, is expected to be *metastable* in the following sense. If two blocks B_0 and B_1 are conflicting, and, say, B_0 happen to receive just slightly more support (i.e. included in just a few more forks than B_1), then B_0 will quickly become *much* more popular while B_1 will be abandoned. What is important here is that even though each node in the network has it own local DAG view and updates arrive in a random order, the preference of B_0 over B_1 becomes evident in all the local views, so it becomes a global consensus that B_0 is canonical while B_1 is abandoned.

In theory, the DAG approach has clear advantages, especially in a highly asynchronous environment.

(NC) There's no competition between miners mining on top of the same block.

[15]The issue whether or not Bitcoin block size should be increased resulted in a deep chasm within the Bitcoin community in July 2017. Two incompatible forks were created on top of block no. 478558. Each fork is now a ledger for its own currency: Bitcoin (BTC) and Bitcoin Cash (BCH).

(INC) Participants even with very few computational resources may non-trivially contribute to the strength of the network consensus (like the total difficulty of the longest chain in the Nakamoto consensus). Thus, the network throughput can in theory scale horizontally with the number of clients.

(FL) The distribution of trust is more flexible: a block may be accepted even if one of the parents is not accepted.

Mining a block (assuming for simplicity it contains a single transaction T) is outilined in Algorithm 4.

Algorithm 4 Mining a block with multiple parents

```
1: procedure MINE($T$)
2:     $tips \leftarrow$ GETTIPS($\mathcal{B}, T$)        ▷ Choose some parent blocks compatible with $T$
3:     VERIFY($tips, T$)
4:     $B \leftarrow$ GENERATEBLOCK($tips, T$)
5:     BROADCAST($B$)
6: end procedure
```

DAG systems, however, are still susceptible to double-spend and Sybil attacks, which are considerably more difficult to tackle than in synchronous networks. Implementing Sybil-resistance while having (INC) is especially challenging: while it should be cheap to participate in the network consensus for a single user, it should be very expensive to flood the network with many cheap transactions and shift the consensus on one's will. Although the research in this direction is very active, it seems that the prevailing line of thought is to somehow integrate either PoW or PoS scheme.

Finality becomes much harder as well[16]: one should be able to decide whether a transaction is final based on a single *local* view, which may be substantially different from the others. Note, that in synchronous systems one can estimate with high accuracy if a local view is lagging behind by analyzing the timestamp of the last accepted block. Rigorous analysis is hard, especially in the presence of adversarial nodes, so current DAG-based solutions use ad-hoc solutions which serve as an anchor to the ground truth: special witnesses (ByteBall)[7], a separate PoW-based snapshot (Vite)[11], or a special "coordinator" node (IOTA)[6].

A notable exception is the SPECTRE protocol [25], which provably satisfies the following properties with arbitrarily high probility using only PoW:

(C) Consistency: no two accepted transactions are conflicting

(S) If a transaction is accepted by an honest node, it will be accepted forever by all honest nodes in the network within reasonable time.

(L) If a transaction is valid, it is accepted by all honest nodes within reasonable time.

Finally, let us mention a plethora of robust pure consensus protocols based on gossiping (communicating with random peers), notably[17] Algorand [24], Hashgraph [16], Avalanche [22], Tendermint [12]. These protocols can in principle be applied on top of any data structure in order to reach a global consensus on e.g. which of two conflicting transactions are canonical. In particular, it might be the case that underlying ideas can be used in order to improve the performance of DAG-based systems.

References

[1] `https://en.bitcoin.it/wiki/Confirmation`.

[2] `https://www.wired.co.uk/article/bitcoin-mining-energy-consumption-new-york`.

[3] `https://github.com/ethereum/wiki/wiki/Proof-of-Stake-FAQs`.

[16] And, strictly speaking, impossible due to the infamous FLP theorem [5]. In practice, one can do rather well without breaking FLP by using e.g. randomness, assuming bounded latency etc.

[17] This list is by no means exhaustive.

[4] https://en.wikipedia.org/wiki/Directed_acyclic_graph.

[5] A brief tour of FLP impossibility. https://www.the-paper-trail.org/post/2008-08-13-a-brief-tour-of-flp-impossibility/.

[6] The IOTA Coordinator. https://domschiener.gitbooks.io/iota-guide/content/chapter1/current-role-of-the-coordinator.html.

[7] A. Churyumov. Byteball: A Decentralized System for Storage and Transfer of Value. https://byteball.org/Byteball.pdf.

[8] V. Buterin. On slow and fast block times. https://blog.ethereum.org/2015/09/14/on-slow-and-fast-block-times/. Accessed: 2018-08-19.

[9] V. Buterin. The problem of censorship. https://blog.ethereum.org/2015/06/06/the-problem-of-censorship/.

[10] V. Buterin. Thoughts on UTXO. https://medium.com/@ConsenSys/thoughts-on-utxo-by-vitalik-buterin-2bb782c67e53.

[11] C. Liu et. al. Vite: A High Performance Asynchronous Decentralized Application Platform. https://www.vite.org/whitepaper/vite_en.pdf.

[12] E. Buchman et. al. The latest gossip on BFT consensus. https://arxiv.org/pdf/1807.04938.pdf.

[13] M. Rauchs et. al. Distributed ledger technology systems: A conceptual framework. https://papers.ssrn.com/sol3/papers.cfm?abstract_id=3230013. Accessed: 2018-08-19.

[14] S. Bano et. al. Sok: Consensus in the age of blockchains. https://arxiv.org/pdf/1711.03936.pdf.

[15] E. Ittay and E. G. Sirer. Majority is not enough: bitcoin mining is vulnerable. *Communications of the ACM*, 61(7):95–102, Jun 2018.

[16] L. Baird. https://www.swirlds.com/downloads/SWIRLDS-TR-2016-01.pdf. https://s3.amazonaws.com/hedera-hashgraph/hh-whitepaper-v1.1-180518.pdf.

[17] S. Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System. https://bitcoin.org/bitcoin.pdf.

[18] D. O'Keefe. Understanding cryptocurrency transaction speeds. https://medium.com/coinmonks/understanding-cryptocurrency-transaction-speeds-f9731fd93cb3.

[19] S. Popov. The Tangle. https://assets.ctfassets.net/r1dr6vzfxhev/2t4uxvsIqk0EUau6g2sw0g/45eae33637ca92f85dd9f4a3a218e1ec/iota1_4_3.pdf.

[20] E. G. Sirer. https://twitter.com/el33th4xor/status/1006931658338177024.

[21] Y. Sompolinsky and A. Zohar. Secure High-Rate Transaction Processing in Bitcoin. https://eprint.iacr.org/2013/881.pdf.

[22] Team Rocket. Snowflake to Avalanche: A Novel Metastable Consensus Protocol Family for Cryptocurrencies. https://ipfs.io/ipfs/QmUy4jh5mGNZvLkjies1RWM4YuvJh5o2FYopNPVYwrRVGV.

[23] V.Buterin and V. Griffith. Casper the friendly finality gadget. https://arxiv.org/abs/1710.09437.

[24] Y. Gillad et. al. Algorand: Scaling Byzantine Agreements for Cryptocurrencies. https://people.csail.mit.edu/nickolai/papers/gilad-algorand.pdf.

[25] Y. Lewenberg Y. Sompolinsky and A. Zohar. Serialization of Proof-of-work Events: Confirming Transactions via Recursive Elections. https://eprint.iacr.org/2016/1159.pdf.