Blockchain Project

Christian Koller Nielsen, 201303527

January 2020

Contents

1 Introduction

Blockchain is the underlying structure in the digital currency Bitcoin and was first introduced by Satoshi Nakamoto in 2008. You can think of a blockchain as a distributed, write-only, immutable memory. It is a continuous growing list of records that is linked and secured using cryptography. Each record is called a block and all blocks are linked together forming a chain. Hence the name blockchain. The goal of this project is to study and learn about blockchain through the DNO book 1.

2 Double Spending Problem

In many distributed systems messages are propagated through an underlying peer-to-peer network. Each node in the network is connected to some subset of other nodes. If you want to send a message you will send it to your neighbor and they will send it to their neighbors and so on. This method of distributing messages is called flooding. This ensures that if you want to send a message, the message will find it's way to spread across the network, even in the presence of a lot of corrupted parties. This is what is known as *eventual delivery.*

Unfortunately this comes with a problem: If you send two messages m_1 and m_2 at the same time, they will reach different nodes at different times. Some machines on the network will see the messages in the right order $[m_1\ m_2]$ and others will see them in reversed order $[m_2, m_1]$. Imagine on peer in the network sending the following messages:

- m_1: Transfer 100$ to pay rent.
- m_2: Transfer 100$ to buy some books

Now if there are only 100$ in my account, then the last message has to be rejected. Hence, it is important which one of the two message was the last one. We need to be able to decide the order of the messages. This problem is called the *double-spending problem* and blockchain offers a solution to this problem without the need of a trusted authority or central server.

3 Nakamoto Style Consensus

Nakamoto Style Consensus (NSC) is a synchronous implementation of totally ordered broadcast (TOB).

Definition 1 (Fully Synchronous System) *Each process has a clock and a known bound of how far they drift away from each other. They is also an upper bound on how long it takes to send messages between any two parties.*

Definition 2 (Fully Asynchronous System) *Processes do not have clocks, so protocols cannot be specified by asking processes to do things at specific times. There is a known bound on how long it takes to send messages. Messages might be dropped or delivered at anytime of an adversary's choosing.*

Definition 3 (Total order broadcast) *A total order broadcast is a broadcast where all correct processes in a system of multiple processes receive the same set of messages in the same order; that is, the same sequence of messages.*

Some criterion's for NSC TOBs are:

- **Autonomy:** No entity can take over the system. To ensure this, make it such that you can join the system without having permission from anyone. We say the TOB is *unpermissioned.*
- **Peer-to-Peer Friendly:** Peer-to-peer networks makes is possible for parties to join the network at free will. This means that communication will be done by flooding messages.
- **Denial-of-Service (DoS) Attack Resistant** If corrupt peers learn someone's IP-address they can flood the person with many useless messages and make so busy that they are unable to answer all legitimate requests. To denial of service attacks NSC protocols do not make the IP of any peers public. If a peer is attacked, it can disconnect and rejoin the network with a new IP.

3.1 Distributed Lottery

NSC protocols work by a distributed lottery. A node wins the lottery in such a way that it can proof it to the rest of the world. By winning the lottery you become the leader and win the right to order the outstanding transactions. The basic idea is that someone can win a lottery and then tell the rest of the network to reorder the outstanding transactions.

In a peer-to-peer setting peers can come and go as they want and there is no agreement on which parties are in the system. Reaching agreement on which parties are in the system is an agreement problem in itself and the problem we are trying to solve is also an agreement problem. Hence, we need to elect a leader without knowing who is for election.

The reason behind the lottery being decentralised is the following: If it was just one server or one organisation saying who won the lottery, it could take over the network and keep announcing itself as the winner. We say that it should not have a *single point of attack*.

The lottery will have the following properties:

- **Local Computation Only:** You do not need to communicate with other parties to participate in the lottery. This way anyone can join without anyone knowing who participates. All you need to know is the current state of the TOB. Imagine a lottery with precisely one winner and a setting where parties can come and go. Peer P_4 won, but it is not there right now. Then there would be 0 winners. Distributed lotteries with local computation must have a random element to them. This means that sometimes there is no winner and sometimes there are multiple winners.
- **Non-predictable:** No one can predict who will win the lottery. Otherwise corrupted parties could DoS the winner and stop it from sending the next block.
- **Public Verifiable:** If you won the lottery, you can prove this to the rest of the peers.
- **One Message**: To prove that you won you send a single message along with the next block. When you send a message corrupted parties learn your IP and then they can DoS you. If they do it after you send your message it doesn't matter.
- **Sybil Attack:** Corrupted parties should not gain influence by creating multiple identities i.e. it should not be possible to enter the lottery for free.

3.2 Tree of Blocks

Since there are multiple winners sometimes, two new blocks are added at the same time and points to the same previous block. Then we would no longer be building a chain, but a tree.

3.3 Longest Chain

Since we have a tree there are multiple chains you can extend on. The rule is to extend on the longest chain you can see. If 51% follow this rule then a long chain will grow with just a few side branches. This is why we need the chain to be sybil attack resistant.

4 Proof-of-Work

Proof-of-work (PoW) is a type of distributed lottery. The idea behind this lottery is that participants have to solve a puzzle. They have to find a random value such that the hash of this value is a bit-string starting with a certain number of 0 bits.

H is a hash-function, A is the last block and $a = H(x)$ is a pointer to the last block. Each of the participants has a verification key vk and counter c. Now participants calculate: $y_c = H(a, vk, c)$. One wins win if the first h bits of y_1 are 0. Otherwise one tries again for a new value of c. One can try as many times as one want. One can adjust how hard it should be to solve the puzzle by changing the value of h.

An attacker who creates multiple accounts gains no advantage in winning the lottery because the chance of winning depends solely on how much computing power one has. The attacker might as well spend all his computing power on one account.

4.1 Energy Consumption

If one wins the lottery you win a fraction of a bitcoin, which currently, is worth a lot of money. If the price of bitcoin goes up people buy more computing power to try and solve the puzzle and win bitcoins. This design wastes a lot of energy and heat. An alternative and more energy friendly type of distributed lottery is called Proof-of-stake (PoS). In PoS each player makes one hash per second and bitcoin currently 10^{20}.

5 Proof-of-Stake

Proof-of-stake (PoS) is done by choosing the next block through a random selection where participants with a higher amount of *stake* have a higher chance of winning. The stake represents how much the user has invested in the system. It could be the amount of crypto cur rency they have on their account. The basic idea is that if one has a lot of money stored in one's bank one tends to have less interest in burning down the bank.

Each account vk has an amount of stake a. The lotteries are run on slots $slot = 1, 2, ...$ The slots are spaced out in time, for example every second. All parties agree when the lottery number $slot$ will be run. To check if vk has won it computes:

$$Draw = Sig_{sk}(\text{"}LOTTERY\text{"}, slot),$$
$$Val = a \cdot H(Draw)$$
$$\text{if } Val \geq Hardness \text{ vk has won.}$$

Since $H(Draw)$ is a random number that is scaled by a, people with higher stake have a higher chance of winning.

To prove that you have won you send $DRAW$ and the receiver checks the signature with vk and checks that $Val \geq Hardness$. This proves that vk has won.

Since the chance of winning the lottery depends on the amount of money you have in total, attackers gain no advantage from creating multiple accounts. This protects again sybil attacks.

The owners of most crypto currency have the highest chance of winning. This makes sense since holders of crypto currency have an incentive to keep the system healthy. If the system is unhealthy currency will go down and they will loose money.

Using signatures protects against DoS attacks because only P_i knows sk_i and can compute $Draw_i$. Thereby no one else can check if P_i won the lottery before he did and start attacking him before he has send out his single message with the next block and a proof he has won. If he gets attacked afterwards it does not matter because he already did his job.

6 Lottery System

Now that we are familiar with the main ideas in PoW and PoS we can go more into detail about how the lottery actually works.

6.1 Slot Time

Each party has a local clock $clock_i$. The slots are spaced out in time such that peers can deliver their messages before the next round/slot starts. There is a value *SlotLength* and the idea is to set it to be big enough for the peers in the network to receive messages before the next slot starts. A party p_i is in slot *slot* if:

$$slot - 1 \leq \frac{Clock_i}{SlotLength} < slot$$

Each party has a draw in each slot $DRAW_{i,slot}$ and for each draw they have a value determining if the win the slot $Val(Draw_{i,slot})$.

6.2 Genesis Block

In the first round the first block is created and this block is called the *genisis block*. Each party generates key pairs (sk_i, vk_i) for a signature scheme, they have a value $Ticket_i$. In PoS this is the amount of crypto currency held by the party. Finally there is a random number *Seed*. This should be unpredictable for all parties before they pick their key vk_i. In the genesis block the public keys, tickets, and the seed are made public.

6.3 Lottery

In each *slot* party vk_i computes the draw: $Draw = Sig_{sk}("LOTTERY", seed, slot)$.
Other parties can verify the draw by checking if is accepting. We write this as $VerDraw_{vk_i}(Draw, slot)$. If $VerDraw_{vk_i}(Draw, slot) = Reject$ then $Val(vk_i, slot, Draw) = -\infty$ otherwise:

$$Val(vk_i, slot, Draw) = Tickets_i \cdot H(LOTTERY, Seed, slot, vk_i, Draw).$$

You win the lottery at the current slot if: $Val(P_i, slot, Draw) \geq Hardness$.

The lottery has the the following properties:

- The *winners are private until announced* because only the party P_i knows sk_i and can compute $Draw_{slot,i}$. As mentioned earlier this prevents DoS.
- The *winners can be recognised* because all parties know $Ticket_i$ and vk_i so all players can compute $Val(vk_i, slot, Draw)$ once they are given $Draw$.
- The *signatures are unique* because otherwise a corrupt party could compute several valid signatures for a winning block and thereby create more chaos.
- The *seed should be made public after the parties have picked their public keys* because otherwise the corrupted parties could pick (vk_i, sk_i) such that $Val(vk_i, slot, Sig_{sk_i}(LOTTERY, slot))$ are as large as possible for future values of *slot* and thereby gain an advantage.

7 Growing a Tree

We got some problem in the current design:

1. Sometimes a winning block gets delivered only to some correct parties in time because the delivery time can be higher than the slot time.
2. Sometimes there is no winner in the slot.
3. Sometimes there are multiple winners in the slot.

The solution to these problems is to grow a tree instead of a chain. Some paths in this tree are better than others. The best path in the tree is a path from the root containing the genesis block G to a leaf that has the longest chain. If some paths are equally long you pick the path with the leaf with highest value.

The following is a protocol for building a tree and choosing which chain to extend:

Building a block Tree

Initially there is a genesis block and the tree only consists of the genesis block. We let the set $S_i = G$

1. At a new slot $slot_i$ all parties do the following:
   - Let $P = BestPath(Tree_i)$.
   - If $Val(vk_j, slot, Draw_i) > Hardness$ then do the following:
     - $B_i = Leaf(P)$.
     - $data_i = ((Received_i \setminus Delivered(P)), Metadata_i)$.
     - $h_i = H(B_i)$.
     - $\sigma_i = Sig_{sk_i}(BLOCK, slot_i, data_i, h_i)$.
     - Send $V_i = (BLOCK, vk_i, slot_i, Draw_i, data_i, \sigma_i)$.
2. On receive V_j all parties do the following:
   - Verify that the signature on v_j is valid: $Ver_{vk_j}(\sigma_j, (BLOCK, slot_i, data_i, h_i))$.
   - Verify the signature on the draw i valid: $VerDraw_{vk_i}(Draw, slot)$.
   - Check that it is a winner: $Val(vk_j, slot, Draw_i) > Hardness$.
   - Store it and wait till the slot is over and then add it to S_i.

7.1 How does the tree grow

The tree grows over time t and not all parties see the same tree at the same time because there is a delivery time Δ_t on the messages. All nodes seen by any honest party will reach them by time $t + \Delta_t$.

Definition 4 (honest tree) *Given two trees the union of them is a tree that - starting from the root - contains all paths in both trees. An* honest tree *is the tree formed by the union of all honest parties' trees.*

By definition all correct parties see a tree which is a subset of the honest tree. If delivery time is small, all honest parties basically see the honest tree and therefore agree on the best leaf.

If there is time enough between the slots and the honest tree does no grow for long enough, honest parties will see the same tree and be on the same path and only have small unlucky branches growing.

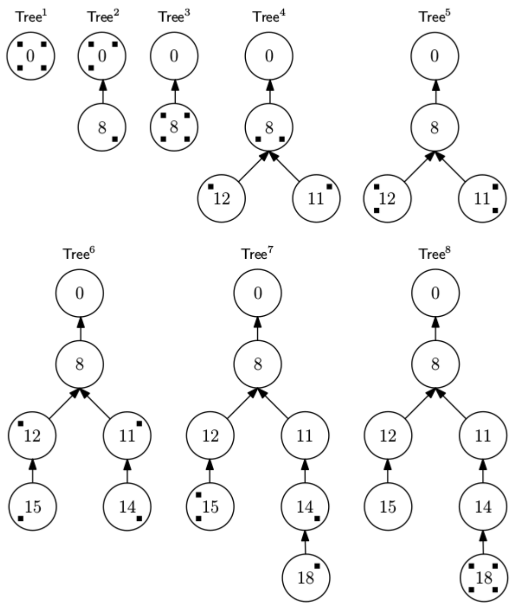

Figure 1: **Tree Growth:** A circle means that some party won that slot number and created a block. A square in a circle means that the block considered the best leaf for a party. We start with a genesis block and then at slot 8 a new block is created that all peers agree on. At slot 11 some party wins and then right after before seeing block 11 another party wins slot 12. This creates a situation where two new blocks reach different parties at different times, creating a tree. Now one party in block 11 wins slot 14 and another party in block 12 wins slot 15 before it has seen the new block. Then at slot 18 we have another winner and there is sufficient time that everyone gets that block and therefore all four parties must consider block 18 the best. Next time there is a block winner they will extend block 18.

7.2 Forking, Rollback and Finalisation

In 1 we saw an example of parties changing from one chain to another. This is called a *rollback*. When a rollback is done, transactions in the shortest chain are potentially lost. Rollback happens when the tree forks. But when does forking happen?

1. Forking happens if blocks are produced faster than they can propagate on the peer-to-peer network.

2. Forking happens when corrupt parties win a lot of blocks.

In case 1 we could have two honest winners who didn't have time to see each others blocks and then they point back to the same block and create a tree.

In case 2 corrupt parties could be interested in creating forks and rollbacks. Maybe they bought a house on the current longest chain. They could extend on the shortest chain and thereby make a new longest chain and make a new truth where the payment for the house is gone. Or maybe they just create chaos and see the world burn, so they keep on extending the shortest paths such that we have two long competing chains. Then you have two long case of things you thought was the truth that can just disappears under you.

The possible rollback length is affected by case 1: The slowness of the peer-to-peer network compared to how fast you produce blocks and case 2: How many corrupted parties there are. This leaves us with the question of when to trust a transaction. We say what some blocks cannot ever be rolled back. Those blocks are called a *final block*. There are two approaches to finding out when a block is final.

1. Wait long enough. Only consider a block when it is up in the tree for long enough.
2. run separate process that detect is the block is final. This we call a *finalization layer*.

7.3 Ghost Growth

To understand how far we need to look back in the tree to find a block that cannot be rolled back, we look at another attack. Corrupted parties can withhold blocks and build a tree. The *ghost tree* is the largest tree that corrupted parties can build with their withheld blocks. The corrupted parties could reveal a branch of the ghost tree that is better than the current best chain and thereby force the honest parties to do a long rollback.

One solution is to say: We do not do long rollbacks. Imagine you are never allowed to rollback longer than 3 blocks back. Imagine if corrupted player now reveal a ghost branch that is 4 longer than the current block. Now the players in the current block are willing to do the rollback, but the players in the head of the ghost branch are not willing to do the rollback. Now agreement will never be reached. On the other hand if you can rollbacks of length 3 and those rollbacks could never happen, then what is the point of banning them?

7.4 Standard Tree Scenario

We assume that:

1. At least 90% of the alive tickets are honest.
2. SlotLength is set high enough that that 95% of all slots are timely.
3. The hardness is set such that 90% of all honest winners are unique, i.e., they win a slot in which there are no other honest winners.

90% honesty is high but it allows us to keep the proofs simple and present the main ideas. 95% of the slots are timely can be ensured by setting the slot time large enough. 90% of all honest slots are unique can be ensured by setting the $Hardness$ such that probability of winning is 10%.

7.5 Three Growth

The tree gets higher all the time as there are more and more winners. About a fraction $TreeGrowth$ of the slot winners make the tree gets higher.

Definition 5 (three growth) *Let $TreeGrowth \in [0, 1]$ be a real number. We say that a protocol has a tree growth of $TreeGrowth$ if after n slot winners the height of HonestTree is very close to $TreeGrowth * n$ except with small probability.*

We say a slot is:

- *timely* if all messages sent in the previous slot were delivered before the current slot started.
- *honest* if there is at least one honest winner in the slot.
- *lucky* if it is honest and timely.

If it is a lucky slot then the tree will grow. An honest party will extend the best chain with their block and their block will reach all parties in time and thus the tree will grow. An adversary might reveal a ghost chain that is longer than the current best chain, but then the revealed ghost chain will grow. When there is an honest winner the tree will grow with height 1 unless:

1. There was already an honest winner in the slot and they both extend the same leaf.
2. The slot was not timely and the winner does not build on the best path in the honest tree because it did not have time to reach the winner.

An honest slot winner is *wasted* if it is not timely or if it had another honest winner with a higher value.

The tree growth for the honest tree at time t will be:

$$Len(BestPath(HonestTree^t)) \geq HonestWinners^t - WastedHonestWinners^t$$

Consider the standard tree scenario. Out of 100 winners on average 90 are honest. Out of these 10% are not unique. That is 9 wasted. Thus 81 are honest and unique. Out of these 5% are not timely. Since 5% of 81 is about 5 we say that is 5 wasted. In total $9 + 5 = 14$ are wasted. Hence then $TreeGrowth \geq 90\% - 14\% = 76\%$.

7.6 Chain Quality

If all nodes on the best path were produced by corrupt parties they can censor some transactions by not adding them to their blocks. Chain quality is a measure of how many blocks on the best path are produced by honest parties.

Definition 6 (chain quality) *A protocol has chain quality of* $ChainQuality \in [0,1]$ *if the best path in* $HonestTree$ *has length* L *then very close to* $ChainQuality *$ L *nodes on the chain were produced by parties which were honest when they produced the node, except with negligible probability.*

Consider the standard tree scenario. There is tree growth 76%, so after 100 winning events the tree grows by at least 76 blocks. So at most 24 blocks sit in side branches to the longest chain. In worst case these 24 are all honest nodes. Of the 100 nodes on average 90 are honest. At most 24 of these were lost to side branches. So 66 of them must be sitting on the longest path. So worst case 66 of the 76 blocks on the longest chain are honest. Since $66/76 > 87\%$ it follows that $ChainQuality > 87\%$.

7.7 Limited Rollback

The *RollBackLimit* is the limit such that the probability that there will ever be a rollback longer than this is some negligibly probability.

- A *super slot* is a slot that was timely and won by one honest party and no other honest parties won the slot.
- A *super block* is the block produced by the party who won the super slot.
- A *filler block* is a block that is not a super blok.

Consider our standard tree scenario. Out of every 100 winners, there tend to be about 90 honest ones. Less than 10% of these will hit a slot with another winner, so there will tend to be 81 that have their own slot. Of these at most 5% will not be timely, so at least $81 - 5 = 76$ will be super slots.

All super blocks sit at different heights in the honest tree. This makes sense since they are won in different slots and they are timely and no other honest parties won that slot.

Lemma 1 (filler slots) *Assume that a rollback is possible from branch* B_1 *to* B_2 *both rooted at node* N*. Since node* N *was created there was* s *super block created, then at least* $s/2$ *distinct filler blocks was created.*

If there is two branches and the super blocks are at distinct heights, then at least half of them must be are not super blocks and thereby they must be filler blocks.

Consider our standard tree scenario. 76 will be super blocks. Therefore there will be 24 filler blocks. Since $2 \cdot 24 < 76$ there will tend to not be long rollbacks because there are not filler blocks enough. From this one can derive the *RollBackLimit* in the standard tree scenario to be: 11062.

8 Finality

Finality has to do with how fast you can trust a transaction that you have put on the blockchain.

8.1 Finality Layer

In the previous chapter we found a limit to the possible rollback length. The finality layer detects when blocks are final oppose to waiting for the us to reach the worst possible rollback length like we did in the last chapter.

8.2 Final Layer Specification

- The genesis block is considered final.
- Once an honest party considers a block final that block will always be considered final.
- If an honest party considers a block final then eventually all honest parties will consider that block final.
- There is a single path formed in the tree which contains all finalized blocks.
- The final chain grows at the same rate as the blockchain.

8.3 Finality Layer Implementation

8.3.1 The finalization Committee

There will be a $FinalityCommittee$ consisting of some of the parties running the blockchain called $voters$. Each voter has a number of votes. This could be the number of tickets. It is important that the committee is mostly honest. We require that:

$$Votes_{corrupt} < Votes_{FinalityComittee}/3$$

8.3.2 Common Prefix

Take any two nodes in the honest tree and walk back c steps in both of them, then the two corresponding positions are on a common chain. $CommonPrefix$ is the smallest number of steps we have to take back that will bring the honest parties on the same chain.

Consider $Three$[7] in figure 1 has common prefix 2. If you start at node 15 and 18 and walk back two steps you are in position 11 and 8. 11 and 8 are on the same path. So 2 is the shortest step back to be on the same chain.

8.3.3 Detect common prefix

A protocol called *DetectCommonPrefix* can be implemented asynchronous TOB. In the protocol all parties input a string x and it either accepts or rejects the string. Since it is an asynchronous system, corrupt players might not answer and therefore we only wait with counting the votes until 67% of the parties have answered. If all corrupt players answer, there could be 33% honest parties we do not hear from. If all 67% vote, then at least 34% of these are honest and at most 33% are corrupt. Therefore we require that 34% vote for x before it is accepted.

8.3.4 One Finalization

Let h be an agreed upon height and Δ be the number of steps to look back in the tree. The height we look at should be higher than any already finalized block. The idea is then to finalize a block at the height h:

1. wait till height $h + \Delta$.
2. B:= The block at height h.
3. vote on B using *DetectCommonPrefix*.
4. if agreement then B is final.
5. else let $\Delta = 2 \cdot \Delta$ and go to step 1.

The idea is that if $\Delta \geq \mathit{CommonPrefix}$ then the voters are are on the same path. Since they start at the same height and go back the same amount of steps, that means that they look at the same block. When they look at the same block *DetectCommonPrefix* will return accept. $\Delta \leq \mathit{CommonPrefix}$ then *DetectCommonPrefix* will reject and we try again at a higher Δ. It terminates because at some point Δ will be bigger than the common-prefix which is guaranteed by limited rollback.

When a node is considered final we remove everything that is not behind the final node. Everything that is behind the final node is considered final. If my car was paid on a branch that was removed that is annoying, but at least I knew it was not final yet. The point is that now you now when something can disappear maybe and when it cannot, because it by algorithm illegal to ever roll back behind a finalized block. Now we know that this part of the tree will never disappear.

8.3.5 Making NSC Finality Friendly

We need to make some changes to the NSC from the previous chapters.

- In each new block we save a pointer to the last final block we saw.
- Do not accept the new block until the last final block it points to is also considered the last final block for the receiver.

- New longest chain rule: Compare the length of the final part of the path and then the total length of the path.

8.3.6 Continues Finalization

Now we know how to do one finalization and we need to know how to pick h and the Δ for the next finalization. At first we pick the next block after the genesis block: $h = 1$ and $\Delta = 1$. If it terminates at height 1000 and then the try next time to agree on $h = 2$ then it will take to long to agree on the current block, because we have to look so far into the past. Instead try to agree a block at height 1000. When a block here is agreed finalized so is all the blocks in between this block and the genesis block. We also don't want to pick h too far into the future because then we have to wait before we can start agreeing on the next block.

8.3.7 Ghost Branches

An advantage to finalization is that ghost branches which lie lower than a finalized block can never be used again.

8.4 Cheating the CAP theorem

The cap Theorem relates three concepts:

- **Partitioning:** A network is partitioned if there are two parts of it that cannot talk to each other.
- **Consistency:** In our case consistency means that everyone has the same view of what is the truth. Do all parties see the messages in the same order? Do we all agree on which block is the head?
- **Availability:** A system is available if it is operational in a timely manner.

The CAP theorem basically says that if a network partitions, you cannot have consistency and availability at the same time.

8.4.1 A Concrete Example

Let's say the network partitions and it is impossible for a day to send messages between America and Europe. Some cable in the ocean was cut or there was a power breakdown or something. There is a guy with 100$ on his account who tries to withdraw all his money. If ATM is available then the guy could walk up to an ATM in Europe and withdraw 100$ and then there would be no way for the system in America of knowing that the his bank account is empty. Thus the system would not be consistent. The guy could then walk up to an ATM in America, withdraw 100$, and walk away with 200$ even though he only had 100$ on his account. Or the system could say we are not handing out money because I don't know what is going on in America and then you lost availability.

8.4.2 What Happens to the Blockchain?

The underlying NCS blockchain would keep on producing blocks on both sides of the cut, so the blockchain remains available. but the committee cannot vote. So finality stops at the last final node, which is positioned before the cut. This means the blockchain will still grow, but then for some time they will see that nothing is finalized. At some point the cut would be repaired again and the messages could parse again and the two chains meet the best chain wins and the shortest one goes away.

The point is that we don't have to choose between availability and consistency on system level - we can dynamically pick both of them.. If we are selling peanuts maybe it is OK to accept a transaction before it is finalized. If we are selling a car we should wait till the block is finalized before handing out the keys.

8.5 Dynamic Block Time

If you produce blocks faster than they can appear on the P2P network you tend to get branching. So you could take some time in between the blocks. That is also why is need to be hard to win the lottery so you can space out the winning. The propagation time can vary a lot depending on whether the blockchain is under attack or not. The faster you produce blocks the more your security will drop and it drops exponentially. If you produce blocks faster than they can propagate you have no security. The block time should be slow in comparison to to the speed of the peer-to-peer network: If you get an attack on the network you want slow block time. If you have a peaceful network you want fast block time. If you look at how far back we looked to finalize then this information we can use to see how much branching there is going on. We can use this as a measure of how healthy the blockchain is. One simple way to do it would be to take the lottery from before and scale the hardness by measure $\Delta * Hardness$. Then we can under good circumstances pump out many blocks and in bad conditions pump out blocks slower but stay save.

9 References

[1],[DNO BOOK], Ivan Damgård, Jesper Buus Nielsen, Claudio Orlandi, ”Distributed Systems and Security, An Introduction to Cryptography, IT-Security, Distributed Systems, and Blockchain Technology”, In: December 5, 2019.