UNIVERSITA' DEGLI STUDI DI NAPOLI FEDERICO II

Scuola Politecnica e delle Scienze di Base
Corso di Laurea in Ingegneria Informatica

Elaborato finale in **Reti di Calcolatori I**

Analysis of blockchain technologies and benchmarking of NXT and Ethereum in emulated network environment

Anno Accademico 2019/2020

Candidato:

Marco Carlo Feliciano

matr. N46003714

Se io fossi un navigatore, avrei
bisogno della stella polare per
orientarmi, per proseguire lungo
la rotta senza perdermi, e del
rum, per ondeggiare leggermente
intorno alla rotta, talvolta
seguendo il mare. La mia famiglia
è per me la stella polare, i miei
amici il rum.
Dedico questa tesi alla stella
polare ed al rum.

Index of contents

Abstract

After Satoshi Nakamoto’s paper « Bitcoin: A Peer-to-Peer Electronic Cash System », blockchain technologies have been spread in many fashions. Bitcoin was just the first one and is still one of the most popular[1], but as the public interest in blockchain technologies grew, efforts were made to develop more scalable, more flexible and more performant blockchain technologies. Developments led to new consensus mechanisms, new types of transaction and even new data structures for the storage of the blocks. We will deal with these technologies in general, then we will cover in detail two interesting blockchain technologies: NXT and Ethereum. After that, we will talk about performance metrics, we will show how we applied some of these to test NXT and Ethereum in an emulated network environment, that is set-up using Mininet, and we will show our experimental results. At the end, we will compare our setup with Hyperledger Caliper’s one, that is a framework for blockchain performance benchmarking.

Chapter 1: Blockchain technologies overview

The first blockchain proposal was intended to be a system for digital payments which didn’t require a trusted third-party[2]. As described in [1], back to the time in which currencies didn’t exist, the only proof of richness was the owning of some scarce goods which was socially acknowledged to be valuable, such as good or salt. Then, there came the long time of middleman-mediated money: from cash to online banks, the problem was that there always was a third-party which controls the creation of new currency. Cash is tangible and easy to steal, but the middleman doesn’t know how much cash does everyone own and doesn’t handle transactions, so it is mostly middleman-free. On the other hand, online currency is intangible and banks are able to commit transactions with the other side of the world within seconds: try to do it with cash; but the middleman is responsible for every transaction and knows the balance of all its users. So, ideally we want a currency that is both intangible and middleman-free: Nakamoto’s paper described a system with these properties, that is a distributed ledger which stores transactions, just like the ledger used in non-digital banks. Following developments led to the creation of blockchain technologies suitable for other tasks than payments, such as voting, smart contracts, and so on: many applications were built on top of blockchain technologies. We will discuss about the basic building blocks of a blockchain, the most common consensus mechanisms, the client’s behaviour related to the CAP theorem, and some advanced features like smart contracts. Then, we will explore NXT and Ethereum further and we will define blockchain performance metrics.

1.1 Introduction

A blockchain, like the name suggests, is a chain of blocks. A block is an element which stores zero, one or many transactions. Each block is chained with adjacent blocks in the chain, which we can see like a linked list (but in general it is a directed acyclic graph), through cryptographic hash functions. Currency is initially generated with a genesis block which states that certain amounts of currency are sent to a set of addresses, then new currency can be generated as block mining rewards like in Bitcoin or can only be redistributed like many other technologies. Redistribution of currency can only happen by means of transactions, which can include transaction fees. There isn't a central authority for the access control of the addresses: each account takes advantage of cryptography to generate a pair (public key, private key), so the address is the public key (or something derived from it) and who owns currency and wants to send part of it must sign the transaction in which he or she sends "money" to an address with the private key, to prove his identity in a way that everyone in the network can check. The transaction is then broadcasted to peers in the overlay network and added in a transactions' pool, waiting to be included in a block. Peers in the overlay network which accept, validate, create and broadcast transactions and blocks are called nodes. Nodes cooperate to keep the blockchain up & running, and each one of them, by running the same node software, follows the consensus mechanism to decide if a block can be added to the blockchain and which blockchain is the right one in case of multiple forks. A client generates and sends transactions to one or more nodes, makes read requests about the blockchain status and decides if a transaction in a block can be considered safe.

1.1.1 Blocks

If we see blockchain as a directed acyclic graph (DAG)[4], then blocks are the nodes, and connections between blocks are the edges. Each node in the graph is responsible for the storing of items called transactions, and each edge is responsible for the consistency of the

DAG. A peculiarity of any blockchain is that every single node is immutable, i.e. you can't change some fields of a node without building a new node (and building a new node requires an effort as we'll see soon later on), and every edge is unique to the pair of blocks it connects, so you can't add a block in the middle of the blockchain: if an attacker wanted to modify a transaction in an old block, then he or she would have to remake all the chain from that old block to the last block chained (see consensus mechanisms). How to achieve these goals? The answer is cryptography[2]. To provide the immutability of every single node, which we refer to as block from now on, there is a cryptographic proof that must be computed. The type of the proof depends on the consensus mechanism, but in general if F is a subset of all fields of a block, N is a nonce and D is a difficulty value for the proof, then the block is valid if and only if the following relationship is satisfied:

$$H(F+N) \leq g(D) \quad (1.1)$$

where H is a cryptographic hash function and g is a generic function of difficulty that depends on the consensus mechanism. The difficulty is adjusted in such a way that the creation of a new block (referred to as "mining" in Bitcoin[2] and "forging" in NXT[3,], for example) requires some fixed time, that is to say that a block is added to the blockchain at regular intervals. This is the effort we have anticipated before: a "miner" has to vary the nonce until it hits a hash that satisfies the relationship 1.1. When it hits a good hash, it adds the block to its local view of the blockchain and broadcasts it to its neighbours in the P2P network. The receiving peers, which we refer to as nodes from now on, check if the relationship 1.1 is satisfied (they can check it because the nonce is included as a field of the block). If the relationship isn't satisfied, the block is discarded, otherwise it is gossiped and the blockchain is updated. Note that if a malicious node tries to modify some of the fields involved in the computation of the proof, then the proof isn't valid anymore and the attacker itself has to compute a new proof, if it is allowed to. We will cover this security concern in detail in the consensus mechanisms' section. To provide the uniqueness of every single edge, the key is still the proof: included in a new block is the hash of the previous block, and this hash is included in the subset F of the fields used for the computation of the proof. So, a blockchain is an append-only data structure.

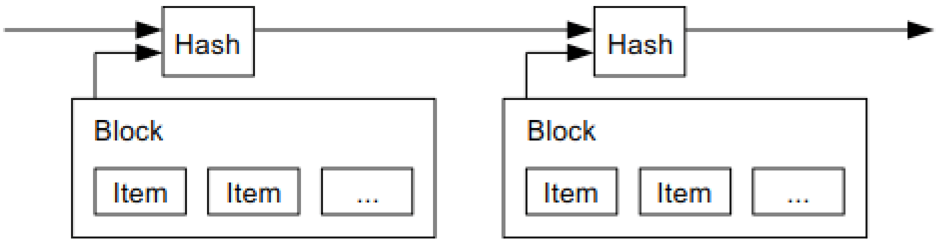


Figure 1: Chain of Blocks (adapted from [2])

A block can store zero, one or many transactions. A block with zero transactions has two purposes: since the difficulty is re-computed after each new block added according to the time that was needed to mine that block, new blocks need to be added even if there are no transactions in the transaction pool, to keep the difficulty consistent with the actual computing power of the network; the other reason is that, for how blockchain works, the head of it is as if it were volatile, i.e. you can't trust the content of the last block because multiple forks could exist and eventually nodes in the network will choose only one of these: so, each block ahead of a given block B of the chain is seen as a confirmation of B. For example, if B is the block with index 80, and the head of the chain has index 86, then B has received 6 confirmations. Mining blocks with zero transactions allows to confirm blocks with one or many transactions. Anyway, there are also other approaches to accomplish this task. We will cover it more in detail in the transactions' section.

Each block, of course, doesn't only store transactions: it also has a header, containing at least its hash, the nonce, the miner's public key, the difficulty at which it was mined, a timestamp, the index of the block in the chain, the number of transactions, the miner's signature of the block, the hash of the previous block, the version of the blockchain application-specific protocol used by the peers to communicate. It is a common practice to also include an ID for the block and the ID of the previous block[3], to simplify read operations on the blockchain (hashes are probabilistically unique, the use of a proper ID enforce uniqueness). Another common practice is to enforce a fixed maximum length for each block, limiting the maximum number of transactions allowed[3].

At this point, the question you may ask is: what about the genesis block?

As we said in the introduction (1.1), currency is initially generated with a genesis block. The genesis block typically has a previous hash of all zeros and is hard-coded in the node software, because it unavoidably violates two rules of the blockchain: it sends non-existing coins and it is chained to a non-existing node. Recent blockchain technologies employ a sequence of genesis blocks, i.e. a "genesis chain", that is all hard-coded in node software. In any case, the currency generated in this way is typically distributed to people who have invested in the development of the system (since many blockchain technologies are open-source and so they might need financial aid). A feature related to the genesis transactions is the block reward[2]: a miner can "print money" by means of mining a new block, receiving an amount of currency proportional to the total number of blocks, in such a way that the reward decreases with time going down to zero. This is one of the incentives that motivate nodes to make sure that the network works well.

The persistence of the local view of the blockchain on a node is usually handled by a DBMS: proof-of-concept and test blockchain technologies use light, relational databases like H2 and SQLite or they just keep everything in main memory, whereas production blockchain technologies use databases like MongoDB, FlureeDB and BigchainDB, as stated in [5]: "… The purpose and design of a blockchain database differ from a traditional database. As mentioned above blockchain is a Distributed Database Management System (DDBMS). This proves that blockchain does not require any conventional database. However, decentralized applications (dApps) that has to handle significant data loads might use a conventional database. In such cases, databases like MongoDB, FlureeDB and BigchainDB are used to enhance the blockchain operability and functionality."

1.1.2 Accounts

Blockchain networks come in many fashions: public and fully decentralized, public and decentralized with group leaders[6], private with access control. Private blockchain technologies have been employed by big banks with many international branches to improve their security and enhance their operations[7]. In the private case, access control is still managed by a logically central authority. In the public case, there isn't a central authority, i.e. there isn’t a database which tracks accounts. You just send coins to an address and there is no way to roll back. Another time, the means to have a public address and a private password to be able to use the coins sent to that address, is cryptography. When you create an account, you don’t register to any site, and when you access to your “wallet”, you don’t submit a password to any form (but there exist client interfaces from which you can see your balance and use on the client-side your private key to make payments and submit signed transactions, see the transactions’ section for more). What you do is generating a pair (public key, private key) with some algorithm like ECC (Elliptic-curve cryptography), according to the blockchain software. The public address is your public key or something derived from it, and the private password is your private key or something that generated it, like a passphrase[3]. If you lose your passphrase or your private key, since there isn’t a central authority, you have no means of gaining access to your account. In some cases, losses or hacking of accounts with very high balance led to the so called hard forks[8], where the clock is turned back through a software update and the system is reset to a previous state, before the hack or in such a way that the balance is forced to be transferred to another account. To make it possible, peers need to agree with the decision: there is a fee offered to them for the help, and still some of them decide to abandon the network and keep running the other fork of the blockchain. Normal users, who don’t have a very high balance, would lose what they own forever. So it is advised to write the passphrase or whatever is used to access the account on *cold storage*, such as a piece of paper or an USB hardware wallet[1], to not lose it and to not expose it to the Internet (everything exposed to the Internet can be hacked). Unlike accounts’ “login informations”, accounts’ balance is usually tracked: nodes keep an up-to-date balance of accounts for performance reasons, otherwise they would have to run across the whole blockchain every time a client wanted to know its balance, for example. So, we told a lie when we said that there isn’t a database which tracks accounts: each node maintains its local database, the same that it uses to store blocks, to store all accounts’ public informations. This is necessary to check if a client is allowed to send a certain amount of coins, i.e. if he or she owns that amount. So, for what has been said, accounts are anonymous: you don’t need to specify your name to create an account. Anyway, governments managed to track accounts to real people[1]: summing up, a pubic key is just a pseudonym, and putting together many pseudonyms and many transactions involving these pseudonyms make it possible to link pseudonyms to web services (maybe illegal, if governments are interested in them) and to real identities. That said, many public blockchain technologies that are not involved in illegal matters let you create an account specifying some informations about you, for the implementation of transaction-based applications on top of the blockchain (one of the first examples is MtG Arena[9]).

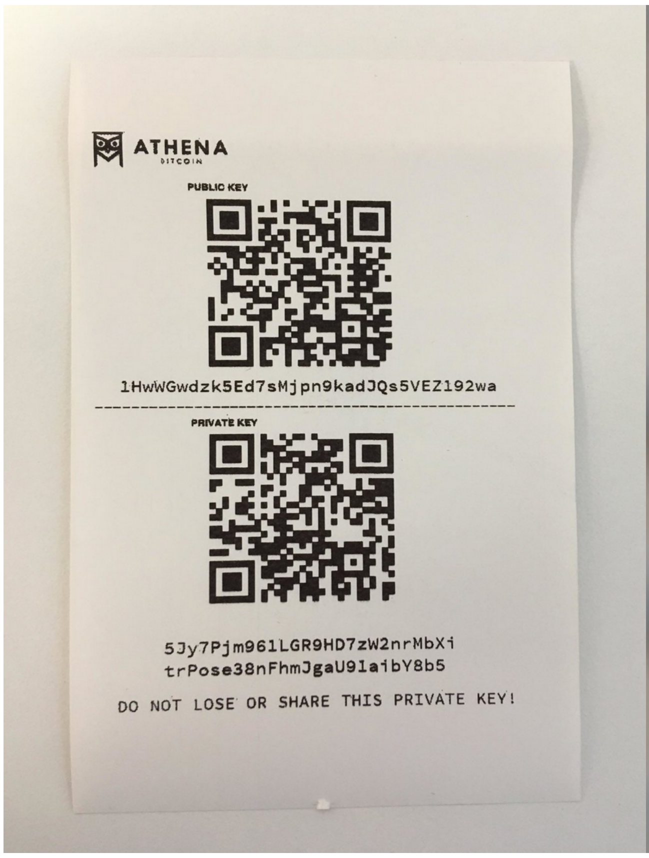

ATHENA
BITCOIN
PUBLIC KEY
1HwWGwdzk5Ed7sMjpn9kadJQs5VEZ192wa
PRIVATE KEY
5Jy7Pjm961LGR9HD7zW2nrMbXi
trPose38nFhmJgaU91aibY8b5
DO NOT LOSE OR SHARE THIS PRIVATE KEY!

Figure 2: A paper wallet (source: Athena Bitcoin)

Figure 3: An hardware wallet (source: Wikipedia)

1.1.3 Transactions

Transactions are the « raison d'être » of blockchain technologies. The entire set of mechanisms regarding security, consistency, authentication and performance of the P2P networks we are talking about is for the processing of transactions. Transactions are both part of the means and the whole purpose of blockchain technologies. In the first blockchain proposal[2], transactions were thought only as ordinary payments. If that is the case, what you send or receive with a transaction can only be something expressed in unit of measures called coins. But further developments led to a more general concept of unit exchangeable in a blockchain network, called token. The introduction of tokens allows to define multiple types of transactions[3], for example to store messages on the blockchain or for account control, or even to create agents that automatically react to transactions sent to them, according to a piece of code called “smart contract”[10] written in a Turing-complete language. We will cover some uses of transactions that go beyond ordinary payments in the Advanced features’ section. We said that blocks store transactions. So, if you run across the blockchain, you can piece together the transaction history. Since each transaction involves tokens sent from an account to another, you can extract each account’s balance from the blockchain: so, if an account attempts to send more tokens than it owns, you are able to mark the transaction as invalid. But, how to prevent someone from sending tokens if he or she doesn’t have access to the account that owns these tokens? In the accounts’ section, we said that you use a private key to gain access to your account, but we didn’t say how. The answer is digital signature. It is the only mean to authenticate yourself and to be sure that nobody else would pretend to spend your tokens: that’s why you hear the term “cryptocurrency” related to the term “blockchain”. When you create a transaction, you have to specify sender, recipient, an amount and you put a timestamp. Depending on the blockchain software, there are other fields such as a message, a transaction fee or a transaction type field, if multiple types of transactions are allowed. In many blockchain technologies, amounts of tokens are just scalar sizes, so you have a certain balance and you can send a certain amount of tokens: you will just add that amount to the recipient and subtract it from the sender. In Bitcoin, objects are used to store amounts, then the owning of currency means that there are Transaction Output objects linked to an account and that this account didn’t spend these objects yet, so they are called UTXO. Each one of these objects has its own hash, which you reference to when you want to spend one or more of them; so, you wrap each UTXO in objects called Transaction Input, you create new UTXOs with the property that the sum of the amounts of the inputs is equal to the sum of the amounts of the outputs, and you put all in a Transaction object.

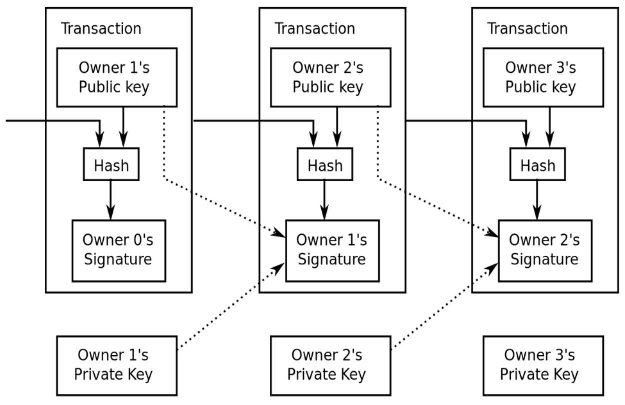


Figure 4: Simplified scheme (from Bitcoin whitepaper[2])

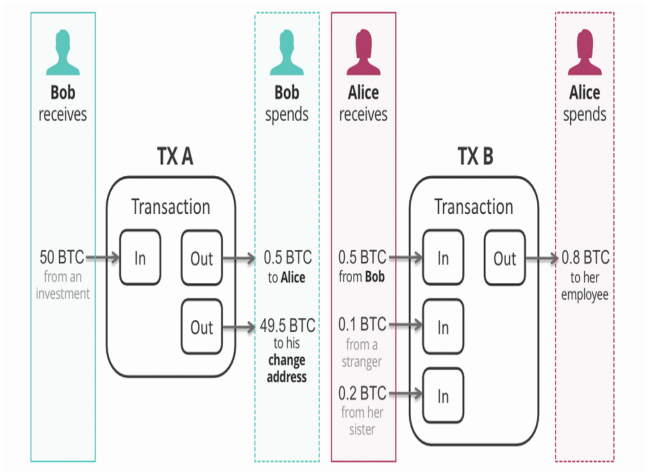


Figure 5: Bitcoin TX example (from [11])

Then, for the authentication, you have to sign the transaction with your private key. The fields involved in the signature's payload have to make sure that a replay attack isn't possible, so for example there is a timestamp and we saw that in Bitcoin there is also a form of cipher block chaining. In some blockchain technologies, the sender is also charged of the burden of computing a little proof of work[6], to avoid the spam of transactions, whereas in other technologies the spam is contained by receiving nodes[3]. The transaction object is then ready to be sent to a node, which validate it and adds it to a list of transactions not included in any block yet; this list is called transaction pool and these transactions are usually called transaction tips or just tips. The node is also responsible for the gossip of the transaction to its neighbours, so that other nodes of the network add the transaction to their pool. A distinguishing feature between different blockchain technologies is the algorithm to choose the tips to include in a block from the pool. If there are transaction fees, nodes will choose the tips in such a way to maximize their profit for approving them by putting them into a block; otherwise, the algorithm's aim is to maximize the performance of the network in terms of TPS (see 1.3). As we said in the Blocks' section, there could be zero, one or many transactions in a block. It is clear that in Bitcoin there is always more then one transaction in any block; some of these are related among them (a payment, its fee and the rest of the money returned to the sender), others are not, they are just included in the same block. We already showed the purposes of a block with zero transactions: but the same purposes can be pursued with a zero-value transaction, if the software allows to. So, we can obtain the same operations that we have with blocks with many transactions using blocks with one transaction each. The withdrawal of the tokens is a negative value transaction, zero-value transactions keep up the blockchain rhythm and positive value transactions are the accreditation. The problem is how to relate a negative value transaction with a positive value one to make an atomic operation that we all know as payment, since different transactions are in different blocks. To implement atomic operations with multiple blocks, blocks can be organized in bundles, and blocks in the same bundle must have explicit links. IOTA uses this mechanism[6]: an overview of it in the figure below, in which links are to be intended as hash chaining between blocks (IOTA uses a DAG).

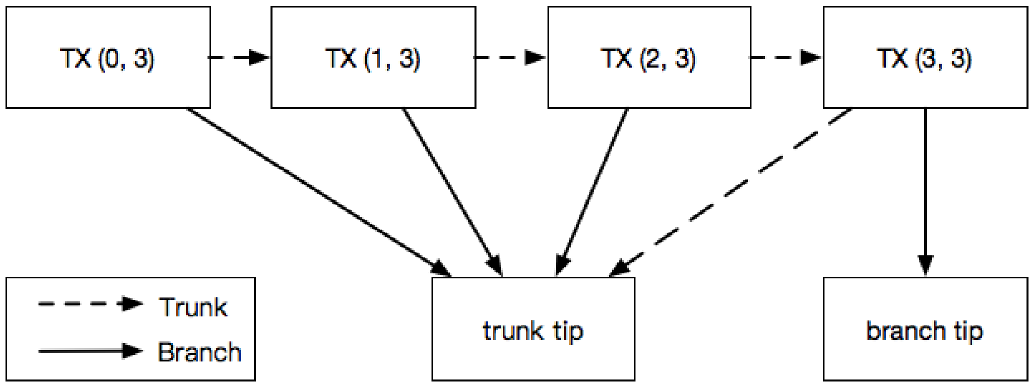


Figure 6: IOTA bundle structure (Source: IOTA docs)

So, for what we have said so far, a single client operation that sends a signed transaction to a node and is almost the same for different blockchain technologies, produces many different behaviours depending from the actual implementation. In other words, while the protocol used by the client to communicate with a node uses a representation of transactions similar for different blockchain technologies, the actual representation of a transaction on the blockchain can vary a lot and can have meanings different from an ordinary payment.

1.1.4 Consensus mechanisms

In centralized networks, all nodes agree on what the central authority says. On the other hand, in decentralized networks there isn't a trusted third party and the network itself is considered untrusted. Each node has a local knowledge of the blockchain and is able to make arbitrary changes to it, but to make these changes effective, they must be acknowledged by all nodes in the network. Nodes in the overlay network communicate among them both at regular intervals and in response to asynchronous events generated by clients. The aim of this communication is to synchronize nodes' knowledge about the blockchain, and the operations include the acknowledgement of the miner of a block (i.e. the validity of a block) and the resolution of conflicts in case of multiple forks (i.e. which chain should we follow if there are multiple valid chains). Eventually, all nodes must agree on the transaction history, so the blockchain must be a single source of truth. In the blocks' section, we anticipated (formula 1.1) that in general a proof must satisfy a relationship between the result of the hash function applied to certain data belonging to the block and the result of a function of difficulty. We also said that the function of difficulty makes the difference among different types of consensus mechanisms. Over the time, many strategies were defined to reach the consensus in blockchain networks, borrowing some of them from distributed systems' theory[12]. Almost any new blockchain technology in production defines its own peculiar consensus mechanism. We will rearrange these mechanisms in three macro-types: Proof of Work (PoW), Proof of Stake (PoS), Delegated PoS (D-PoS). In all the cases, a spread strategy is to fix a mean amount of time that must pass from one block to the following, to let enough transactions accumulate in a block and to let the block run across the network; note that if there is one transaction per block, you don't have to let transactions accumulate and so you can add blocks to the chain at an higher frequency; another two reasons for the fixed amount of time are the limiting of the network traffic, i.e. the P2P network is more likely to fail if the underlying network is congested, and the other reason is the security, as we'll see soon.

In PoW, the function of difficulty of the formula 1.1 produces a result that is the same for all the nodes, or in other words it doesn’t have other parameters outside of difficulty. So, each node creates a payload with the fields of the local block eligible to be mined (after having put in the block its public key and the transactions it chose to confirm) and a nonce, and vary the nonce until it hits an hash which satisfies the relationship 1.1. The problem is that different nodes have different computing powers, so adjusting difficulty in order to fix the time leads to an ever increasing difficulty if each node is motivated to be the first to mine a block, for example because of the presence of block rewards and transaction fees. Then, difficulty grows, nodes become competitive, people behind them invest money in dedicated hardware (mining rig, ASIC[1]) and gather them in mining pools, and the decentralized network becomes more centralized[1]. In addition to that, a lot of computing power and electric energy is spent in useless computations. Furthermore, scalability is poor because little nodes will not join in aid of the network because they don’t have an incentive. So, what is the point in using such a consensus mechanism? The point is that the network is available and secure. It is available because there are many mining pools trying to gain profit by mining blocks. To explain why it is secure, we have to show the other side of the consensus, that is to say which chain peers follow in case of multiple forks.

$$validChain = argmax\left(cumDiff\left(chain.lastBlock()\right), chain \in forks\right) \quad (1.2)$$

The valid chain is the one with the highest cumulative difficulty. The *cumDiff* function extracts the cumulative difficulty from the last block, which can be either just the index of the block, so the valid chain will be the longest one, or a cumulative sum of the difficulties of the blocks mined so far, and in that case the valid chain will be the one with the “most effort” spent. Note that this formula applies to blockchain technologies with linear structure, but it can be generalized to technologies with a DAG structure: in that case, we don’t speak of longest chain, but we speak of heaviest tree[12]. Now, about security, the first attack to which we can think of is the so called double-spending attack[2]. An attacker sends an amount of coin to an address to buy something; the recipient then sees the transaction and the balance updated, so the owner of the address ships what the attacker paid for.

Meanwhile, the attacker, in its local view of the blockchain, starts mining blocks and listens for incoming blocks. In the first block that the attacker mines, there is a transaction where he or she sends the money to an address owned by the attacker itself. So, the same money is spent two times: the attacker first spend it to receive a good from a vendor, then it moves its money to an address that it still owns, maintaining the control of it. Then, the attacker stops mining blocks when the blockchain he or she generated is longer than the blockchain recognized by the network.

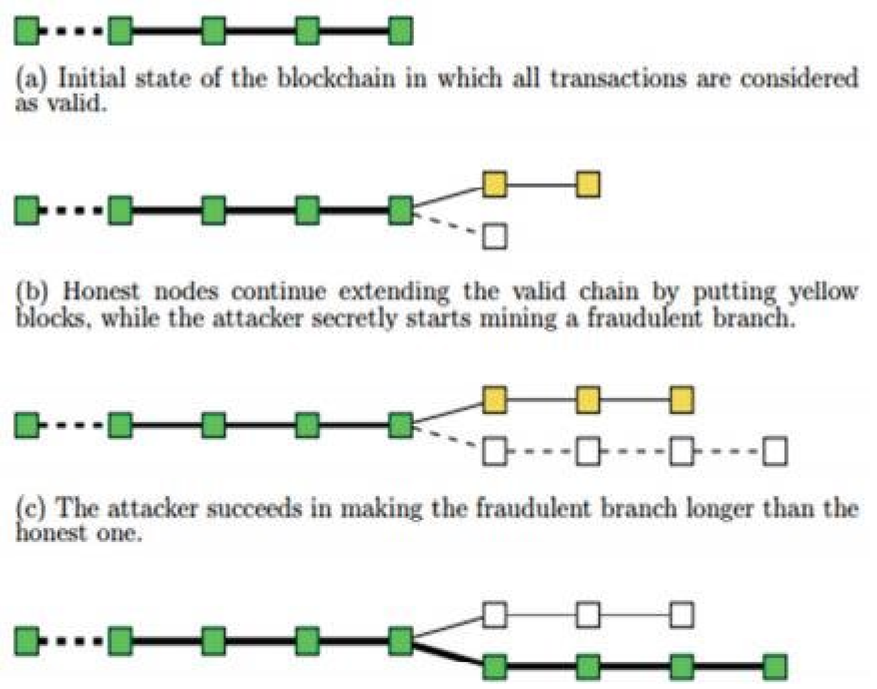

(a) Initial state of the blockchain in which all transactions are considered as valid.
(b) Honest nodes continue extending the valid chain by putting yellow blocks, while the attacker secretly starts mining a fraudulent branch.
(c) The attacker succeeds in making the fraudulent branch longer than the honest one.

Figure 7: Double spending attack (Source: ResearchGate)

But, for what we have said about PoW, nodes are competitive about the mining of blocks, so an attacker would never succeed in this attack unless it owns at least 51% of the computing power of the network. For this reason, the double spending attack against a blockchain that employs PoW is known as 51% attack. An attacker who owns this power would gain more money by block rewards and transaction fees than what he or she would gain from fraudulent transactions. So, this consensus mechanism, coupled with block rewards and transaction fees, is an incentive for nodes to behave honestly. For the same reason, like we anticipated in the Blocks' section, immutability of the blockchain is guaranteed. A more detailed analysis in [2].

In PoS, the function of difficulty of the formula 1.1 takes the miner's data as parameter.

This data includes at least its balance as a scalar unit, but for example the approach used in Peercoin[3] takes as additional parameter the so called coin age, that is the time in which a certain amount of money remained stationary to an account. In general, the balance alone, or the product between balance and coin age, is referred to as stake. The greater is your stake, the greater is the result of the function of difficulty, and so the mining, which can be more properly called forging in this case, is easier, i.e. the proof requires less time to compute. Then, holding a stake is like owning mining rigs: this is why the effort done for computing the PoS is referred to as virtual computing[12]. One could argue that this still leads to centralization: PoW leads to centralization around computing power, PoS leads to centralization around the richest, because they are more likely to forge blocks. Enforcing all tokens to be distributed in the genesis block, therefore removing block rewards, mitigates the problem: even if a node tries to get a lot of tokens from tips, it will eventually spend them because it's pointless to own a lot of tokens without using them. When the node spends the tokens, the virtual computing power is redistributed, so the network remains quite decentralized. Blockchain technologies in production proved that these considerations are true: Bitcoin, that is PoW, risked many times to fall in the hands of a single mining pool[1], whereas PoS technologies like NXT[3] never came up close to these risks. A security concern about PoS is the nothing at stake attack: since the computing power required for mining is far less compared to PoW, a malicious node could work on many forks of the chain in parallel, trying to maximize its profit and at the right moment to perform a double spend attack. To protect against this attack, there are many strategies. One of the most effective[10] is to enforce nodes who wish to validate transactions to place a security deposit: a forger node who produces something invalid or behaves dishonestly loses the tokens it deposited and its ability to participate to the consensus mechanism. This is the technique employed by Casper, one the consensus mechanisms used by Ethereum.

In D-PoS, as explained in [13], there is a form of digital democracy, since anyone can be delegated for the forging of a block. Typically, there are 21 to 101 eligible delegates, called witnesses, whose probability to be elected is proportional to their stake. More precisely, as described in [12], a verifiable random function is used for the voting process, and a witness' probability to be voted is higher if its stake is higher. A delegate has an incentive for being available for the forging because of block rewards. Using this paradigm, there is an high degree of performance and decentralization. In fact, the fixed amount of time between consecutive blocks can be kept low thanks to the election mechanism, (about 2 seconds, whereas Bitcoin's fixed time is 10 minutes). Anyway, if an elected witness doesn't forge a block in time, it loses reputation and can be replaced. There are many implementations of D-PoS: the sharp dividing line between them is the presence of the deposit. Like the Casper mechanism, a deposit is an amount of tokens sent to a secure address, which let you join the consensus mechanism and motivate you to behave well, otherwise you lose your tokens. So, if you want to implement your own D-PoS mechanism, consider the presence of the deposit as an option. We said that delegates are usually 21 to 101: the minimum is large enough to ensure decentralization, the maximum is bound due to network speed limits. In fact, one of the problems of D-PoS, together with the difficulty of the implementation in production compared to PoS and PoW, is the network traffic. The election mechanism produces messages which must be spread all over the overlay network to make sure that every node acknowledges as elected who claims to be a delegate. So, the volume of traffic is bigger than other consensus mechanisms, and is even bigger if the number of witnesses increases: to keep up the pace with the fixed amount of time, messages should travel fast enough.

Other consensus mechanisms are described in [12]; we'll see some other techniques to reach the consensus, or coupled with an existing consensus mechanism, aimed at improving one of scalability, decentralization or security at the cost of other one of them in the Advanced features' section.

1.1.5 Clients and CAP theorem

Clients are the actors who generate events that change the state of the blockchain. In fact, since we said that transactions are the « raison d'être » of blockchain technologies, we can say that clients are the main actors of these systems, because they create transactions and submit them to nodes. Of course, it isn't required to a client to have a deep knowledge about the underlying technology, so user-friendly interfaces based on HTTP are often provided, and furthermore these interfaces, alongside to the interfaces between peers in the overlay network, are implemented using REST services[14]. But a client has also a wired logic in its software: when it fetches the last few blocks of the blockchain and sees a transaction involving itself, for example a payment that it received, it must decide whether to consider this transaction reliable or not. In effect, the logic followed by the client to make this decision is the same logic followed by the nodes. In fact, a client which just has a view of the last few blocks, say 100 blocks for example, is called a lightweight node, whereas a client which stores all the history of the blockchain and accepts requests from other nodes is called a full node[1]. So, how clients make decisions? The answer is quite short, but it has many implications on the features of the P2P network, so we will first introduce something useful for the analysis of these implications: the CAP theorem.

As explained in [15] and [16], CAP stands for Consistency, Availability and Partition Tolerance. The theorem states that a system can’t have more than two out of these three core requirements. A service that is consistent operates fully or not at all: more precisely, every read request receives the most recent write or an error. For example, on your favourite e-store you can add an item to your basket, but if this item is the last in stock, and another user wants to add it to his or her basket, then a consistent system would let just one of them add the item to the basket, and would send an error message to the other one. A system that is not consistent would let both of them add the item to the basket and proceed to payment: if you scale it up to thousand of transactions, it is an huge issue. A service that is available is a service that responds to requests without errors: more precisely, every read request receives a (non-error) response, without the guarantee that it contains the most recent write. Partition tolerance, as defined by Gilbert & Linch (the ones who proved the CAP theorem, that was formulated but not proved by Brewer): “No set of failures less than total network failure is allowed to cause the system to respond incorrectly”. So, a service that is partition tolerant is one which continues to operate despite an arbitrary number of messages being dropped (or delayed) by the network between nodes. You have to decide if you want to drop consistency, availability or partition tolerance. Dropping partition tolerance is not an option in distributed systems, because the logic and the load is distributed and the data is replicated on many different nodes, by design. So you have to choose between consistency and availability. If you drop availability, it impacts negatively your business, because for example if an user enters his or her credit card details and the server comes back with an HTTP 500 error instead of redirecting him or her to a proper page which acts as an handler of the error, then the user would probably be likely to shop elsewhere and very likely to phone his or her bank. On the other hand, we said that the lack of consistency can also be an huge issue. But actually, it is not necessary for a system to be immediately consistent: many workarounds lead to systems that are eventually consistent. In the example where two users want to buy the same item that is last in stock, a workaround could be to let them proceed to payment, and then the conflict would be resolved by making the second order become a back-order. Another subtle point is that a system can't have the three core requirements *at the same time*: it means that in some situations it can be AP, in other situations it can be CP. Now, how the CAP theorem applies to blockchain technologies? Consider the theoretical, unrealistic attack illustrated in [17]. An attacker wants to double spend its coins without mining itself a side fork, but making sure that nodes in the network will mine the side fork for him or her. So, the attacker sends money with a transaction to a vendor, with a certain transaction fee and submitting the transaction to a node that is close, in terms of overlay network, to the vendor's client. Meanwhile, the attacker double spends its coins, submitting the transaction related to the double-spending to many nodes that are far from the vendor's client, and including in this transaction an higher fee. Nodes would include the transaction with the highest fee in the block mined, and so they would create a fork different from the one created by the node that is close to the vendor's client: because they are many nodes, they mine blocks faster than it, so eventually the coins will be double spent. If the attack somehow was successful, there would be at least two different situations to consider. The situation in which the vendor's client immediately considers the coins as sent to it and therefore owned by its wallet, is the AP case. In that case, the success of the attack and the eventual consistence of the blockchain in which everybody acknowledges that the coins were not sent to the vendor, lead to a fraud. For a CP scenario, the client must wait for other blocks to be added on top of the blockchain before trusting the payment: in this way, even if the attack succeeds, the fraud isn't committed. In practice, clients of blockchain technologies in production trust a transaction only if it is in a block that is X blocks behind the head of the blockchain. We already anticipated in the blocks' section that each new block on top of the blockchain is seen as a confirmation of the previous blocks' content: so, requiring that a transaction is trusted only if it is in a block that is X blocks behind the head means that a transaction requires X confirmations. This remark allows us to define in which way a transaction is reliable even if the blockchain is a DAG: each block reachable from a starting block going forward through edges is a confirmation of the starting block itself. The point is that a blockchain technology behaves in a certain way regardless of the client's configuration, but the fact that the system is AP or CP depends by the client's implementation.

1.1.6 Advanced features

Although blockchain technologies were conceived as payment systems, following developments led to the use of blockchain technologies as infrastructure for decentralized applications. To allow a more general use of these technologies, the first step is to define a set of primitive types of transactions[3]. For example, there could be defined transactions to send arbitrary bytes of (encrypted) data from an account to an another, in this case the blockchain would act as a database with a cryptographic access control. Another example is the possibility to create polls and votes, and, alongside with it, the possibility to assign aliases to account, as if the blockchain was a kind of DNS. Decentralized applications can be built on top of blockchain either using ordinary payments as tokens, or taking advantage of the different types of transactions, which make the development easier. Anyway, there is a deeper approach in which the decentralized application, or at least part of it, is executed by the blockchain nodes themselves: this is the approach proposed by Ethereum[10]. This approach involves executable entities called smart contracts. A smart contract is a piece of (compiled, assembly) code that is executed by a lightweight virtual machine. Each node is able to execute that virtual machine because it is in its node software and, moreover, the instruction set is Turing-complete. To create a smart contract, you use a transaction in which you embed the assembly listing of the smart contract; you don’t have to write code in assembly: the most spread smart contracts’ programming language is Solidity, which can be compiled to Ethereum VM’s bytecode. After the transaction is included in a block, the smart contract is created and ready to be executed. When you want to execute a smart contract, you send a transaction to it, calling one of its functions. The minimum amount of tokens required for a smart contract’s function to be executed depends on the blockchain technology and on the computation that must be made, but typically there are some operations, like read operations, that can be done for free. The tokens’ requirement is there because you want to: avoid spam, motivate nodes to execute the smart contracts, force the application to behave in a transaction-based fashion. Smart contracts themselves can call other smart contracts after being called: this leads to security concerns and led to real attacks, as explained in [18]. As an example of what a contract can do[10], a single contract can simulate an entire currency. Another use case for smart contracts or primitive transaction types is the reputation system. Blockchain technologies are born as a reliable way to do transactions in untrusted networks. You can’t have more than two out of three among security, decentralization and scalability: this is a trilemma.

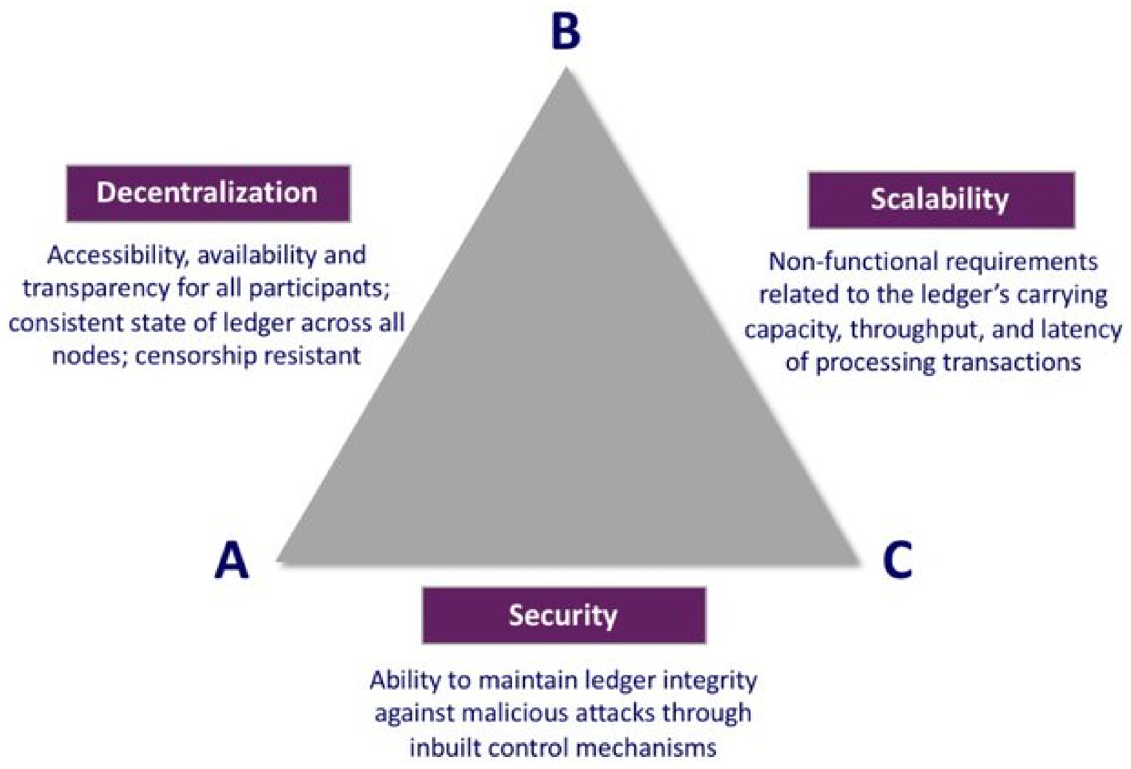

B
Decentralization
Accessibility, availability and transparency for all participants; consistent state of ledger across all nodes; censorship resistant
Scalability
Non-functional requirements related to the ledger's carrying capacity, throughput, and latency of processing transactions
A
C
Security
Ability to maintain ledger integrity against malicious attacks through inbuilt control mechanisms

Figure 8: The blockchain trilemma (Source: [19])

The main challenge is improving the scalability[1] while keeping the decentralization. In fact, private blockchain technologies improved scalability, but they are no longer decentralized. There are blockchain technologies which are decentralized with group leaders to achieve a faster consensus[6]: this approach improves scalability, but is both less decentralized and less secure because a group leader can be a single point of failure. There are many examples of group leaders in blockchain technologies: one it IOTA's Coordinator, which is a client which sends trusted transactions called milestones; developers tried to replace it with a more decentralized system to achieve the same goals, with an action called "The Coordicide"[6]. We started this discussion from the use case of reputation system: now we can link the discussion to that. A more decentralized alternative to group leaders is the employment of trusted nodes, which is flexible because a node can become trusted if other nodes say that it behaves well and can lose its reputation if it doesn't behave well any more. Logically, it's like having group leaders, but these leaders are not hard-coded nor managed by a central authority. The reputation system isn't only for scalability: it can potentially improve security. Anyway, there are blockchain technologies in production whose nodes wouldn’t accept big changes, because there is mining business around them, like Bitcoin[1]. The improvement of the scalability of Bitcoin is a complicated challenge, and it is made even more complicated by the miners. So, some workarounds were proposed and used. One example is the lightning network. Let’s call two users who commit transactions between them Alice and Bob. Consider the situation in which Alice sends 2 BTC to Bob, then after a certain amount of time Bob sends 0.5 BTC back to Alice, and so on. Eventually, Alice and Bob will have a certain balance. Without the lightning network, each one of these transactions requires to be added in a block, and for the CP behaviour of clients, some time must be waited before considering the transaction reliable. As explained in [20], the lightning network is a dedicated channel in which Alice and Bob commit transactions, using signatures. In practice, Bitcoin has a scripting language that allows transactions to specify arbitrary complex conditions for spending outputs. The lightning protocol, which is a “layer 2” protocol upon the blockchain, works as follows. Alice and Bob creates an initial transaction, called funding transaction, in which each one of them puts an equal number of bitcoins. The transaction has a single output with a two-signature condition: output coins can only be spent in a transaction signed by both Alice and Bob. Then, they create another transaction, called commitment transaction, where what they put initially comes back to them. But they only submit the first transaction to the blockchain. If they want to make other payments (or even micro-payments), they create another commitment transaction, which replaces the previous one. For example, if they initially put 5 coins each, there is a total of 10 coins, and the initial commitment transaction states to send 5 coins to Alice and 5 to Bob. Then, if Alice wants to send 2 coins to Bob, then the new commitment transaction will state to send 3 coins to Alice and 7 to Bob. This process can be iterated an unlimited number of times, allowing for secure and fast payments in private channels. Each user has the possibility to “cash out” whenever he or she wants, submitting the commitment transaction to the blockchain. Neither Alice nor Bob is ever in danger of having their coins stolen by the other party. The problem which catches the eye is that if Alice and Bob only wanted to make one single transaction, they would still need to make two transactions, and so there are even more transactions to submit to the blockchain. But another feature of the protocol is the possibility to securely go through other channels to make a payment: if Alice has a payment channel with Bob and Bob has a payment channel with Carol, then Alice can pay Carol by sending some money to Bob and asking Bob to forward the money on to Carol. This technique is known as payment routing. The protocol guarantees that Bob won't be able to steal the money. In this way, thousands or even millions of payment channels can be kept open to form a single global payment network: to reach new recipients, you go through many channels. All of this, with just a relatively little number of transactions on the underlying blockchain, related to cash out. Benefits are granularity (you can make micro-payments that you wouldn't be able to do as standard transactions due to transaction fees[1]), privacy, speed, transaction throughput.

1.2 Blockchain implementations

In this paragraph, we will dive deeper into two blockchain technologies: NXT and Ethereum. The first is interesting because, at the time of its launch, it claimed to be the "descendant of Bitcoin" by its developer[21] and was one of the first blockchain technologies to employ proof of stake; in addition to that, to avoid the using of "scripts" that were in vogue in Bitcoin, NXT defined a set of primitive types of transactions to provide advanced features. The second is interesting because, as we anticipated in the Advanced features' section (1.1.6), it proposes smart contracts and therefore it aims to be a distributed application platform[1].

1.2.1 NXT

We will follow [3] in this discussion about NXT. NXT represents one of the first blockchain technologies to propose solutions at Bitcoin's known issues, such as non-scalability, low TPS rate, and "centralization" resulted from PoW (peers with low mining power either leave mining because they will never get a reward or join a mining pool, making it stronger, and the result is that the top five mining pools gathered together sum up to the 70% of the total mining power, making the system somewhat centralized). In addition to that, NXT adds a degree of flexibility that is not reached with Bitcoin scripting languages, through primitive transaction types. The aim is to let developers implement different types of applications upon the blockchain not limited to applications based on payments. Examples are the support to the storage of little chunks of encrypted data, digital asset market, voting system, multi-signature transactions, additional security mechanisms such as penalties for nodes which don't behave well, and so on. The total amount of tokens available in the network is one billion, and they were all distributed through the genesis transaction, following the logic of a fund raising to state how much tokens to send to which address. The very first transaction left the genesis account with a negative balance of one billion: for this reason, everybody can freely access to that account and furthermore, sending tokens to that account is like burning these tokens. Note that we speak of tokens and not of coins: this precisely by virtue of the fact that NXT is suitable for many applications. It is not possible to generate new tokens, but to redistribute them, there is a transaction fee for each transaction included in a block, and of course the redeemer of the fees is the node which successfully chained the block, according to the consensus mechanism. The consensus mechanism is proof of stake. Differently from Peercoin's mechanism, NXT's one is not based on coin age, that is to say for how much time tokens remain stationary as credit of an account, but the election's probability of a stakeholder is calculated according just to the account's effective balance (we'll see later the meaning of "effective") and to other parameters that are the same for all accounts: time from the last block and the so called base target value. In each block it is stored a value called cumulative difficulty: this value is used to derive the base target value of the following block and to implement a "longest chain" consensus mechanism based on the total effort spent for forging blocks. Balance used in the election, which we called effective balance, is the amount of tokens that remained stationary in a block for at least 1440 blocks; this is useful to prevent an attacker to manipulate probabilities of election moving tokens from an account to another. Furthermore, to avoid the tampering of the chain starting from the genesis block, and in general to avoid the tampering of "old" transactions, the modifiable section of the chain is a sliding window of the last 720 blocks:

this is in addition to the “longest chain” consensus in case of multiple forks. So, in case of multiple forks, nodes make a decision, but there is a “fixed checkpoint” that must be common for all forks, otherwise they are discarded. From the client’s point of view, a transaction is considered reliable only if it was included in a block that is at least 10 blocks behind the head of the blockchain. The process of “mining” blocks in NXT is called forging due to the virtual computing power provided with PoS. Difficulty in forging is adjusted in such a way to generate a new block about every 60 seconds. The proof of the block generation is called block generation signature. Again, formula 1.1 applies: to win the right to forge a block, accounts compete, trying to generate an hash value that is less than a target value. You start from the base target value, which is function of the cumulative difficulty of the previous block. Afterwards, each account computes its target value like that:

$$T = T_b * S * B_e \quad (1.3)$$

where T is the new target value, T_b is the base target value, S is the time since last block, in seconds, and B_e is the effective balance of the account. Note that target value increases with time and is higher if higher is an account’s balance. Cumulative difficulty is computed like that:

$$D_{cb} = D_{pb} + \frac{2^{64}}{T_b} \quad (1.4)$$

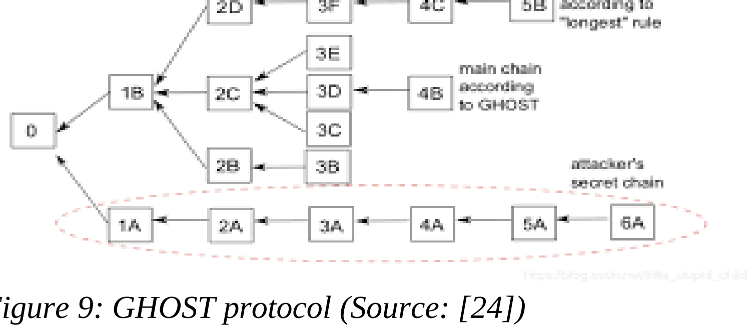
where D_{cb} is the difficulty of the current block, D_{pb} is the difficulty of the previous block, T_b is the base target value for the current block. At this point, the forging algorithm works as follows: each “active” account cryptographically signs previous block’s generation signature, then applies SHA256 function to the result, obtaining what is called account’s hit. If the generated hit is less than the target value, then the block can be generated. There isn’t a nonce: you wait some time to make target value higher and you compare the already computed result of the SHA256 with the new target value. You are “faster” at forging if your effective balance is higher: no mining pools, please. This explains what we previously said about effective balance: it protects against an attack known as shuffling attack. Each device which joins the forging process or sends transactions, therefore having the NXT software in execution, is called node. Nodes belong to one of these types: normal and hallmarked. An hallmarked node is tagged with an encrypted token derived from an account's private key, like a digital signature. This token can then be decoded to reveal the specific address of an NXT account and its balance. The fact that a node is hallmarked adds a degree of "accountability & trust": the higher is the balance of an hallmarked node, the higher is the trust you can give to that node. An attacker wouldn't try to gain trust using hallmarking because the barrier to entry (that is to say the amount of tokens needed to gain trust) is high. Each node is able to process and to gossip in broadcast both transactions and full blocks. Every field of a block sent in broadcast are validated: therefore if it turns out that a node sent in broadcast a block with some invalid fields, to avoid the propagation of invalid data in the network that would only cause useless traffic between nodes, that node could be inserted in a temporary blacklist. Another blacklisting mechanism is the integrated DDOS protection of the overlay network: each node limits the number of requests that it serves from other peers to 30 per second. Just like other cryptocurrencies, transactions' history is stored in a distributed ledger that is a linked chain of blocks. An account can be unlocked only with its private key. In order for an account to generate blocks, it must be unlocked and it must also be active. To be active, its effective balance must be greater than zero: in this way, it is eligible as a forger. Each block has a maximum dimension of 32 KB and can contain up to 255 transactions, each one of these with its header, plus the header of the block. The header of the block contains: version of the NXT application-level protocol, height value (index in the chain), ID of the block, timestamp (seconds from the genesis block), ID and public key of the account who generated the block, ID and hash of the previous block, number of transactions stored in the block, total amount of NXT tokens related to transactions and fees in the block, payload length (the payload is the list of transactions), payload's hash (usually computed as a Merkle tree, as described in [2]), block generation signature, signature for the entire block, base target value, cumulative difficulty. Wallets are integrated in NXT's design: an account is generated starting from a passphrase, with the following pipeline: passphrase →

private key → public key → account ID → visible account number → account address. The account address is the one that starts with “NXT-”, such as “NXT-2543-6FUN-HS5W-BNVW”. Therefore, the key for the unlocking of an account is the passphrase. Wallet’s management allows for the visualization of various types of balance, such as effective balance, basic balance and forged balance, that is the balance obtained by forging blocks, for example. The only mean of modifying an NXT’s account balance, or in general an NXT’s account state, is through transactions. Each transaction performs only one function, that is witnessed by the block in which the transaction is included. The minimum transaction fee is 1 NXT, but a client can urge an higher priority in the elaboration of some transactions by including an higher fee. Each transaction is considered unconfirmed until it is included in a valid block in the blockchain, which is like to give a confirmation of the transaction, as we explained in 1.1. Each unconfirmed transaction in the transaction pool has a deadline: after that deadline, that is 24 hours by default, the transaction is removed from the pool and will never be processed. A transaction can remain unconfirmed for three reasons: it is invalid, it is malformed, it is preempted by transactions which offer an higher transaction fee. NXT’s modular development is possible thanks to the organization of transactions in types and subtypes. There is a total of five types: payment, messaging, colored coins, digital goods, account control. For example, a subtytpe of account control is the “effective balance leasing”. We said that there don’t exist mining pools with the NXT’s consensus mechanism. Anyway, effective balance leasing transactions allows an account to lend its effective balance to another account, until the expiring of a timer: this allows for the creation of forging pools, where accounts involved in the leasing share their profit from transactions fees. Some subtypes of messaging are: arbitrary message, alias assignment, poll creation, vote casting. Arbitrary messages can be encrypted and can be useful to store structured data such as JSON to trigger services developed on NXT infrastructure. Alias assignments allow for the implementation of an alias system, where you can permanently assign an human-friendly string to an account, in such a way to implement URI on blockchain to identify decentralized services: it’s like to have a DNS for NXT accounts. Poll creation and vote castings allow for online voting and surveys.

More advanced features come from the combination of arbitrary messages with colored coins: an example is the NXT Multigateway, which allows to transform Bitcoin, Litecoin and other cryptocurrencies in NXT assets. NXT’s software scales well on portable devices and techniques are being developed to reduce, in the future, the dimension of the blockchain: this is called blockchain pruning. NXT showed to be robust against attacks like nothing at stake and history attack, thanks to effective balance and fixed checkpoints.

1.2.2 Ethereum

Like explained in [10], back in 2013 there was much interest in labelling coins in such a way to represent real world objects as digital assets: efforts were made to implement features like colored coins, decentralized exchanges and financial contracts upon blockchain technologies. Moreover, there was interest in “decentralized autonomous corporations”: it is a mean to handle relationships among investors, owners and employees directly on the blockchain, without the mediation of a legal contract. These organizations were difficult to implement without allowing arbitrary complex computations upon the blockchain. Ethereum’s aim is to create a Turing-complete cryptographic, distributed ledger to encode arbitrarily complex contracts that will be mediated entirely by the blockchain. Therefore, whereas other blockchain technologies aim to build features, Ethereum is “feature-free”: it should be as simple as possible with respect to the implementation of smart contracts’ model, and any other feature is implemented by the participants of the technology who create and publish smart contracts on the blockchain network. We anticipated something about smart contracts in the Advanced features’ section in paragraph 1.1. Now we will dive deeper into them, and in general into Ethereum’s mechanisms. This time we will start with the consensus. Ethereum’s consensus mechanism evolved over the time[22]. Recall from 1.1 that, in blockchain technologies, consensus has two aspects: decision about the confirmation of a new block on the chain, decision about the valid chain in case of multiple forks. In Ethereum, the first aspect is the one which evolved, whereas the second remained stationary. So, we will focus on the second aspect first. Many blockchain technologies adopt the strategy of the longest chain, many others like NXT use cumulative difficulty. The problem with both these strategies is that the effort spent by nodes following a valid, honest fork is wasted if another fork is then selected, and this is not a rare situation. So, if the mining or the forging process is intended to make any node contribute to network security, then the situation previously highlighted shows that there is a security flaw: an attacker wouldn’t compete against the whole set of the honest nodes, but only against the mining pool (or the single node) with the greatest hash power. In addition to that, there is a centralization issue, because the pool with the greatest hash power would mine most of the blocks and will take all the block rewards. The blocks successfully mined and wasted are called “stale blocks”. Bitcoin reduces the number of stale blocks by setting an high time between two consecutive blocks (ten minutes), but if you want the block time to be shorter, you have to do something else[23]. Ethereum uses the GHOST (Greedy Heaviest Observed Subtree) protocol. GHOST includes stale blocks (or Uncles as Ethereum calls them) in the calculation of which chain has the highest cumulative difficulty. This mitigates the network security issue; to also mitigate the centralization issue, stale blocks’ miners are given 87.5 % of the standard block reward, and the miner of the main chain block receives the remaining 12.5% of the standard block reward for each uncle block included. There are many rules regarding the uncles, but for our purposes we only say that the Ethereum version of GHOST only goes down seven levels in the height of the blockchain to include stale blocks. As stated in [24], advantages of this approach are the possibility to increase the block speed without having security flaws, the solution of the problem of the lone block reward and the proof that to increase the TPS rate (we will define it in 1.3) you can’t just blindly increase the block size. So, there is a gain in scalability and in security.



0
1B
2D
3F
4C
5B
main chain according to "longest" rule
2C
3E
3D
4B
main chain according to GHOST
3C
2B
3B
attacker's secret chain
1A
2A
3A
4A
5A
6A

Figure 9: GHOST protocol (Source: [24])

The other aspect is the decision about the confirmation of a new block on the chain. At first, since Ethereum's aim was to be simple, it used a proof of work consensus mechanism, but different from others. In fact, it was discovered that a valid proof of work function should require not only a large number of computations, but also a large amount of memory: it should be memory-hard. Therefore, Ethereum used an algorithm called Dagger, which respected this property of memory-hardness by using moderately connected directed acyclic graphs (that's why Dagger: DAG-ger). Dagger's algorithm required an estimated of 512 MB of RAM per thread, then it made possible for ordinary computers to compete against specialized hardware, because memory is more expensive to produce than SHA256 hashing chips: RAM used in ordinary computers is already essentially optimal[25]. But, as described in [22], this PoW mechanism was successfully attacked in 2017, because it doesn't work securely in a network with a low market cap[1]. Ethereum's long term goal is to implement a proof of stake consensus mechanism with deposit, whose name is Casper; it was intended to be released in January 2020, but it needs enough deposit to be effectively deployed: it will maybe happen in 2021[26]. Meanwhile, Ethereum's developers needed to fix consensus problems faster. They developed a solution easy enough to implement yet effective enough to fix the net. This solution is the Clique protocol, that is a proof of authority (PoA) scheme. A PoA scheme is based on the idea that blocks can only be forged by trusted signers. Given a list of trusted signers, a block eligible to be chained is gossiped to the network, and all the trusted signers (or a quorum, it depends on the implementation) must cryptographically sign the block in order to make it added to the chain. In Ethereum, just one signature is required, but given a list of N authorized signers, any signer can only confirm 1 block out of every K, to limit the damage that would be provoked by a malicious signer. In this way, other signers can vote out the malicious one in the other blocks. In addition to that, to avoid blocks' censorship made by a malicious group of signers who attempt to not being removed from the authorization list, the allowed blocks' confirmation frequency is one out of N/2: the malicious signers need to control at least 51% of the signing accounts to checkmate the honest signers. To protect against malicious signers spamming votes and racing for the confirmation of new blocks as soon as they are allowed to, there is a window of W blocks in which vote proposals are considered in the voting process (outside of it, votes are not summed up) and each signer adds up a small random offset to the time it releases a new block. Ethereum forbids cascading effects: if a signer is dropped and after that drop a previous vote casting reaches consensus, resulting in a drop or in an addition of a signer because of the decreased number of signers required to approve a proposal, then there is a cascading vote. Forbidding this cascade is good for security. Another feature is the possibility for ordinary users to configure proposals on the signers: they can use block headers to say something like “add/drop this signer to/from the list”; there can be many proposals per block, and a signer randomly picks one of these proposals for each block it signs and injects it as a vote. We will see the Ethereum’s block header soon, but for now what we have to know is the presence of these three fields: extra-data, miner, nonce. Extra-data is often used to put client and version of miners, but it can also be used for alternative messages. Miner and nonce are obsolete in a PoA scheme, so they can be re-used. In addition to that, whereas it is a pain to change, add or remove blocks’ header fields in a protocol, it is no pain to change fields’ length if the encoding doesn’t assume a fixed length, that is the case of Ethereum, as we’ll see soon. Therefore, the extra-data field is used to keep track of the signers, so you can look at an old block and verify the signatures, for example, without separately storing the history of trusted signers. This is enough for validation, but another problem rises: we need to update a dynamic list of signers. We said that miner and nonce fields can be re-used in a PoA scheme since they are obsolete, and in fact they are used for voting. In regular blocks, both fields are set to zero. If a signer wishes to make a change to the list of authorized signers, it will set the miner field to the address of the signer it wishes to vote about, and set the nonce to 0x00...00 to vote in favour of adding it to the list of signers or to 0xFF...FF to vote in favour of kicking it out from the list. To enable a fast read of the actual list of trusted signers without having to compute all votes from the genesis block, other PoW fields such as epoch of ethash can be re-used: epoch transitions act as stateless checkpoints containing the list of authorized signers. Now, we’re done with Ethereum’s consensus mechanisms. Back to the fact that Ethereum is a decentralized application platform, you may wonder why Ethereum needs a currency. As anticipated in Advanced features’ section in 1.1, tokens are required to execute smart contracts as fees, and the main reason is to avoid spam. In fact, without execution fees, a smart contract could be a logic bomb, leading to a distributed denial of service in the blockchain network. Ethereum’s currency is called ether, and the most little unit of ether is called wei: in fact, 1 ether = 2^{64} wei. The hard cap of currency is of 2^{128} units, that are not distributed at all with the genesis block but are issued as block rewards. All data in Ethereum is stored with a recursive length prefix encoding, to serialize array of strings of arbitrary length and dimension into strings, and this allowed the switch from PoW to PoA: without it, Ethereum would have needed an hard fork to change consensus mechanism. A full block is stored with a block header, a list of transactions and a list of uncles (recall from GHOST protocol). The block header contains the parent hash, the hash of the uncle list, the coinbase address of the “miner”, the state root (that is the root of a Merkle tree containing (key, value) pairs for active addresses and smart contracts in the blockchain, more in [10]), the hash of the transaction list, difficulty, timestamp, nonce, extra-data and other data related to consensus protocol. Transactions don’t point to TX input/output objects like in Bitcoin, but instead scalar balance is used for accounts and smart contracts; to avoid replay attacks, nonces are used in transactions, and the nonce used in a transaction is the number of transactions already sent by the address signing that transaction. In addition to that, transactions contain data items, which are, in most cases, EVM’s assembly code. All transactions in Ethereum are valid: invalid transactions or transactions with insufficient balance simply have no effect. A fee must also be specified in the transaction, in order to pay the nodes which include that transaction in the block and possibly execute a smart contract. The contract creation transaction is a transaction sent to the zero address; in order for the contract to be created, the fee must satisfy this relationship:

$$tx.fee \geq newContract.fee + memoryFee * tx.numberOfDataItems \quad (1.5)$$

Note that there is a general fee for new contracts, and a fee for the storing of each data item. In fact, the EVM has a total of 256 volatile registers and a virtual permanent memory with 2^{256} entries, which constitute the permanent state of the contract. There are many opcodes: some of them let a smart contract interact with other smart contracts, some of them are just load and store operations, some of them are cryptographic operations. There is a different fee for each class of operations. To activate a smart contract, you send ether to it, and if the number of steps required to execute the smart contract is less than 16, then the fee is zero, otherwise you compute the fee, and each step increases the total fee. The total amount of fee to execute a smart contract, or in general to process a transaction, is called gas. A smart contract can also make requests on the Internet: for example, a smart contract can be used in combination with a weather website. In that case, the weather website is called oracle. For obvious reasons, the oracle fee is higher than other fees, and this is critic because many smart contracts require the access to an oracle to be "smart". Note that, according to the design principles of smart contracts[10], a smart contract can exist for an arbitrarily long time and can have arbitrarily many participants: therefore, it is available and its scalability is, de facto, the scalability of the blockchain network. Moreover, smart contracts can create other smart contracts themselves, because everything is a contract. In real life, smart contracts were used for many purposes, for example to implement stablecoins and to implement good ideas that never took flight because of their complexity, such as Namecoin[1]. The Ethereum protocol doesn't support many features like multi-signature transactions, multiple inputs and outputs and so on that even Bitcoin provides, but instead all complexity comes from a Turing-complete assembly language: it can do anything that any cryptocurrency will ever be able to do.

1.3 Performance metrics

We will summarize [27], adding some little considerations.

Typically, distributed systems are designed in such a way to boost global performance, splitting threads of work among various nodes, a practice that is called load balancing. This is often not true for blockchain technologies: adding more nodes results in enhancing a property of the distributed system called resiliency, that is the capacity to recover a consistent state in situations in which there are inconsistencies or multiple valid consistent states, and to be available in situations in which some nodes go down. But this enhancement in resiliency unavoidably leads to performance losses, caused by redundant checks made by nodes to reach consensus. Therefore, reaching an high degree of performance while keeping the property of resiliency is a challenging task. Our purpose is to define effective performance metrics, but to do that we have to go through some definitions first. The performance evaluation of a system is about the measures of performance of that specific system, according to some performance metrics. On the other hand, although it is often confused with performance evaluation, the benchmark is the process of repeating standard measures to compare a system with another one. The test harness is the set of hardware and software used to conduct the experiment, and is typically composed of different clients which inject workloads to the nodes, soliciting them to perform operations, while observing traffic generated and state of the system. Clients can be present at different levels of the system, but the most important distinction is between load-generating clients and observing clients. A load-generating client has the responsibility of sending transactions, for example: it has to simulate the behaviour of the whole set of users of the system. An observing client make measures on the system, both performing read operations and capturing traffic or monitoring the CPU and memories of the system, depending on its position in the test environment. Both types of clients typically use the standard interfaces provided to users to inject workloads and perform read operations: for example, it’s much easier to see how many transactions were validated in a block using the HTTP API instead of analysing the traffic. The SUT (System Under-Test) is the whole set of hardware, software and networks required to execute the blockchain. Each node is an independent computing unit which communicates with other nodes in the network, that is to say that each node is a virtual entity which cooperates with other nodes to confirm transactions. An important parameter to hold in consideration is the network size, for what we said at the start of this paragraph, intended as the number of nodes connected among them in the overlay network. The last observation before defining the metrics, to better motivate them, is the fact that you can see a blockchain like a finite state machine. If a blockchain is a finite state machine, then a transaction is a state transition. When a client sends a transaction and it hasn’t been confirmed yet, namely a tip transaction, it is like a signal that the finite state machine receipts from one of its sensors: it can ignore it or it can queue it and eventually change state as result of that signal or do nothing, discarding it. The block that contains transactions is like the CPU of the finite state machine, and this CPU instructs the machine’s actuators to do the operations required to change state. The machine also provides some interfaces to make sure that its state can be measured. Now we can define some key metrics.

- Read latency: it is the difference between the time in which a response is received and the time in which the client made a read request; an example of read request made often is the request for the last block, and this is done not only by clients, but also by nodes, to implement their consensus mechanism.
- Read throughput: given an observation time, call it total time, the read throughput is the total number of read operations on the total time; for this reason, it is expressed as RPS (Read operations Per Second). It can be useful as information but it isn’t the main performance measure of blockchain technologies, because it is common practice in production to use dedicated systems for read queries.
- Transaction latency: this is a shared view of the overlay network of the time that was needed for a transaction to be considered confirmed and available in all the network. The latency goes from the time in which the transaction was sent by the client to a node, to the time in which the result of the processing of the transaction is largely available in the whole network, i.e. the finite state machine has successfully performed a state transition. Therefore, the measure includes propagation time of the transaction inside the overlay network and convergence time of the consensus mechanism. To hold in consideration both these factors, the latency must be observed from the point of view of each node of the SUT. Furthermore, the metric is computed for each transaction, so it makes sense to aggregate data using some statistical indicators like minimum, maximum, mean and

variance.

- Transaction throughput: it is like the read throughput, but for transactions, therefore it is expressed as TPS (Transaction Per Second). It is perhaps the most important metric, and it is the ratio at which valid transactions are confirmed by the blockchain SUT in a given observation time; note, again, that it is not the ratio from the point of view of a single node, but is about the global view of the overlay network.

At this point, we have to define how to state that a test is “correct”. In order for a test to be considered correct, it is mandatory to record all environment parameters and the software used for the tests. This recording includes for example CPU and memory allocated to each peer, topology of the network, number of nodes, consensus mechanism, observing client’s behaviour (CP or AP?). These characteristics are, to all effects, part of the test results: making these details available renders easier to compare the results of the same performance tests executed on different platforms. Ideally, observing clients should be external to the SUT, but when this is not possible, it must be recorded and the observing points internal to the SUT must be specified. Also, the characteristics of the transactions used as workload must be recorded: for example, a transaction which contains many data is transmitted slowly in the network. The workload must be designed in such a way to be similar to the real use in production of the blockchain technology. Another test case can contain faultloads instead: since in blockchain technologies invalid and evil transactions, spam and other attack vectors are on the agenda, it can be meaningful to create situations like these, to emulate real faults that happen. Another example, which doesn’t depend on the blockchain logic but depends on the underlying network infrastructure, is the fall of a link between two or more nodes in the overlay network, i.e. the partitioning of the network. An experimental performance metric is the blockchain work. It is defined as a function of the transaction throughput and the network size, which we recall as the number of nodes connected among them in the overlay network. The aim of this metric is to express in the most complete way the work done by the blockchain network in a single figure. Consider for example a blockchain network with a single node: this system could have a very high throughput, but it doesn’t provide guarantees of availability and integrity. Therefore, blockchain work provides a meaningful characterization of blockchain performance considering the distributed system as a whole, rather than simply measuring the performance of what can be seen logically as a database. An ideal function hasn’t been discovered yet, but a suggested one is the following:

$$blockchainWork_{experiment} = TPS_{experiment} * \log_2\left(networkSize_{experiment}\right) \quad (1.6)$$

Chapter 2: Performance tests

We will describe the basic Mininet design and workflow, our test suite and the topologies we built with mininet for our test suite. Then we will detail some aspects of the implementation of the tests, such as the behaviour of the load-generating client and of the observing clients, some details about the API and the configuration of NXT and Ethereum, and we will show and compare the results. At last, we will compare our test setup with Hyperledger Caliper's one.

2.1 Setup of an emulated network environment for blockchain testing with Mininet

We will follow [28] for the introduction to Mininet.

2.1.1 Introduction to Mininet

Mininet is a system which allows for rapid prototyping of large networks, even on the limited resources of a single laptop. It was born as a tool to prototype software-defined networks (SDN). In contrast with other network emulation tools, which use a VM per switch/router and a VM per host, Mininet uses lightweight virtualization. Each host and each switch is a shell process with its own network namespace, obtained in Linux with the *unshare* system call. Therefore, each Mininet node has one or more virtual Ethernet interfaces, created and installed using *ip link add/set* and a pipe to a parent process that spawns all Mininet nodes, that is "mn". A link is just a virtual pair of Ethernet interfaces:

packets sent through one interface are delivered to the other. You can easily create a network with two hosts and two switches connected in a line with this command:

```
sudo mn --topo linear,2
```

Then, you have a CLI to interact with hosts and switches. You can, for example, try to ping one host from another, or set up a simple HTTP server on an host and make requests from another host. Thanks to the lightweight virtualization, Mininet allows for an high degree of scalability, enabling the prototyping of complex networks with hundreds of hosts on a single laptop. If you suddenly have a world-changing idea about a new network protocol, you can evaluate it on Mininet very fast, without losing too much time for setting up a prototyping environment yourself. This is also because Mininet provides a Python API to create custom experiments and topologies: a few lines of code are required to create custom topologies; then, you just have to decide which commands to send to hosts to run your experiment. Another characteristic of Mininet is the fact that it allows for rapid prototyping of SDNs: it has the native support for it. Everytime you start a network with mininet, an SDN controller is spawned, and in fact you can verify that there is OpenFlow traffic on localhost. This is the architecture:

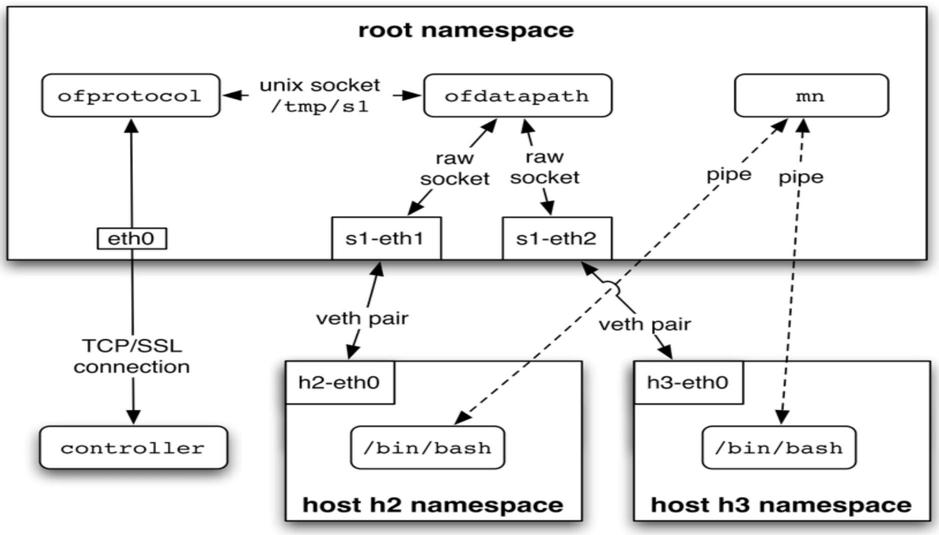


Figure 10: Mininet network architecture on the OS (Source: [28])

There are real examples of optimizations of SDNs that were made possible by the use of Mininet. Furthermore, another example is Ripcord, that is a modular and extensible platform for creating scale-out data center networks. Mininet helped in: creating a common codebase, testing and deploying it to hardware. How? Mininet comes in a bundle: in fact you can download the VM with Mininet (it is an Ubuntu image) and all of its dependencies pre-installed without the needs for additional configurations, download the software of the experiment and execute it on your VM. In addition to that, to deploy on real hardware you just have to configure real hosts and real switches like hosts and switches on Mininet, because Mininet allows for realistic experiments; you don't have to deploy the controller: with SSH servers on hosts and OpenFlow datapaths on switches, you can interact with the network deployed on hardware in the same way you interacted with the emulated network.

In syntesis, Mininet provides a prototyping workflow with the following attributes: it is flexible, deployable, interactive, scalable, realistic and share-able.

2.1.2 Test suite and related topologies

The test suite we created for blockchain technologies can be aggregated in three macro-tests, each of these performing many tests, varying some parameters. The first one aims at only measure CPU and RAM usage and amount of traffic between peers of many nodes connected in a star topology that do nothing else than forging blocks in a fixed period of time of 180 seconds, so there isn't any load generating client and the observers are the nodes themselves. The second and the third ones aim at measure the same things of the first one, plus transactions throughput, expressed in TPS, and total aggregated traffic on peers' interfaces, with respect to the number of transactions sent in a fixed period of time of 200 seconds; instead of showing TPS and total traffic as functions of offered TPS, we showed them as functions of the number of transactions, recording the fixed period of time as an environment parameter. We will then use it to interpret results. The second and the third test cases both have a tree topology, with hosts as leaves, the load-generating client connected at the root of the tree, and for each switch connected to hosts, there is another switch connected to that switch, with two observing clients connected, i.e. there is an observing client per host and these observing clients are added in such a way to divide broadcast domains. The third test case is different from the second because in the time interval [50, 100] seconds the link between the root switch and its right child goes down, resulting in a network partition by half; so its aim is to evaluate how blockchain technologies react to this partition. In addition to that, because the second and the third test cases have two varying parameters, we provide as output parametric plots, with TPS on transactions and the number of peers as a parameter. This is the formal test suite definition:

Test ID	Description	Preconditions	Input	Output	Postconditions
1	Star topology, 180 seconds, number of peers in range [5, 10 … 40]	N/A	Start forging, stop forging	Graphics and spreadsheets for CPU, RAM and traffic with respect to number of peers	Nodes' databases updated
2	Tree topology, 200 seconds, tree depth in range [1,2,3], number of transactions in [250, 500 1000, 2500, 4000, 5000, 10000]	N/A	Start forging, transactions and read operations, stop forging	Graphics and spreadsheets for CPU, RAM, traffic and TPS with respect to number of peers and number of transactions	Nodes' databases updated
3	Tree topology with link up and down, 200 seconds, tree depth in range [1,2,3], number of transactions in [250, 500 1000, 2500, 4000, 5000, 10000]	N/A	Start forging, transactions and read operations, stop forging	Graphics and spreadsheets for CPU, RAM, traffic and TPS with respect to number of peers and number of transactions	Nodes' databases updated

Table 1: Test suite

An example of the tree topology described, with a depth equal to two, is the following:

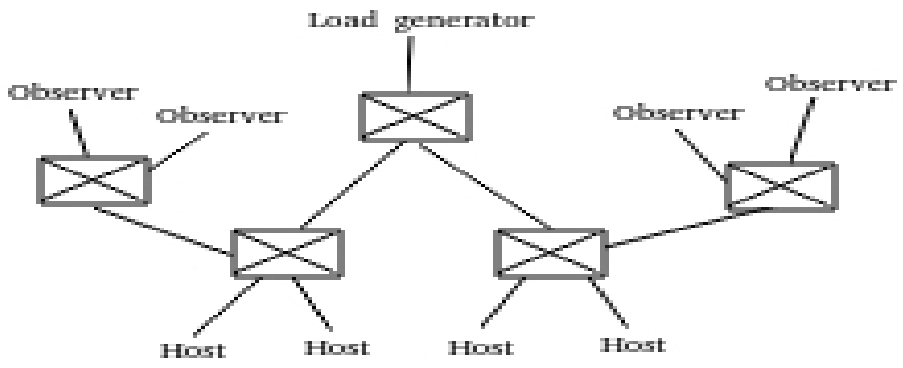


Figure 11: Tree topology of the SUT

2.2 Testing blockchain technologies

VISA banking circuits are capable of processing thousands of transactions per second[1]. Whether a blockchain technology pretends to be a currency or not, transaction throughput is an important factor. Furthermore, with emerging technologies like these of IoT, it is also important for nodes' software to be scalable to little devices. Lastly, another indicator of performance of a P2P protocol is the bandwidth required to a peer to join the overlay network; this is easier to interpret with respect to the global amount of traffic, because the former depends on too many factors, such as the topology of the overlay network. We'll show some of our experimental results about these factors, for NXT and Ethereum. Full results can be found at the following link:

https://drive.google.com/drive/folders/1qhZWsXX98hWjWK2OKtvIjP4jRGAbK2Hz?usp=sharing

2.2.1 Implementation of the tests

The starting point is the recording of the environment parameters that were not explicitly written in the test suite. These include the configuration of the links of the Mininet network to simulate the presence of a WAN, the CPU and RAM allocated to the Mininet VM, the hard disk technology and amount of space available, the behaviour of the clients. We will include everything except the behaviour of the clients, which requires additional discussion, in the following table.

VM system specs	Number of CPU cores	1
	Threads per core	2
	CPU clock frequency	2.3 GHz
	CPU model	Intel Core i5-6200U
	RAM allocated	2048 MB
	RAM block size	128 MB
	RAM type	DDR4
	RAM configured speed	2133 MT/s
	Hard disk type	SSD
	Hard disk size allocated	10 GB
	Hard disk measured throughput	3.1 GB/s
Mininet hosts' interfaces specs	Medium delay	50 ms
	Standard deviation of delay	10 ms
	Distribution of delay	Gaussian
	Correlation between consecutive packets	25,00%
	Probability of loss	10,00%
	Probability of packet loss burst	65,00%
	Probability of bit corruption	0.01%
	Probability of reordering	10,00%
	Probability of duplication	5,00%

Table 2: Environment parameters

Note that we are considering a system with modest hardware specs, except for the hard disk throughput, which is fine to be high because many cheap devices for IoT, like Raspberry Pi, make use of flash memories. So, VM system specs can be compared to these of a device like Raspberry Pi, and therefore we can check if nodes' software is scalable to tiny devices. About network parameters, we supposed a modest network congestion, enough to observe some TCP retransmissions in Wireshark when scrolling traffic capture files. We set these network parameters as described in [29] and [30], that is to say using "tc" Linux utility for traffic control. Now we have to describe the behaviour of the clients. The load generating client sends transactions in a round-robin fashion to blockchain nodes, in bursts of 50 transactions, repeating that for a number of bursts that depends on the total number of transactions, as the sleep time between two consecutive bursts. The number of bursts is the total number of transactions on the number of transactions in each burst. The sleep time between two consecutive bursts is the total time on the number of bursts. Since the total time is fixed to 200 seconds, if for example we have a total of 5000 transactions, then the number of bursts is 100 and the sleep time is 2, i.e. there is a burst every 2 seconds. Note that the sleep time is uniform, and so is the number of transactions in a burst. Anyway, if there are 5000 transactions in 200 seconds as input, i.e. 50 transactions every 2 seconds, then the offered TPS load is 25. We will need this consideration later. The fact that the load generating client sends transactions to nodes in a round-robin fashion is a simulation of the actual clients' behaviour: each client can be connected to a different node, it wouldn't be realistic to send all transactions to the same node. It is up to the blockchain technology to reach consensus. Below is a code snippet taken from an Ethereum test case showing what we said so far:

```
target = "http://{}:8501"
interaction = """curl -m 2 -X POST -H "Content-Type: application/json" --data '{}' """
for i in range(n_bursts):
    print("Transaction burst number {}".format(i+1))
    for j in range(tx_burst):
        host_num = j % len(hosts) + 1
        host_IP = hostToIp('h{}'.format(host_num))
        payload = interaction.format(interact_json(host_to_ethAddr("h{}".format(host_num)),
"0xfb9a175032adbd79e54258cd1b4ce87f8b17e8aa")) + target.format(host_IP)
        try:
            net.nameToNode['h{}'.format(number_of_hosts + 1)].cmd(payload)
        except:
            print("One transaction not sent to h{} due to some  exception".format(host_num))
        elapsed = time.time() - start
        if elapsed >= total_time:
            stop_it = True
            break
    if stop_it:
        break
    time.sleep(total_time // n_bursts)
```

We didn’t include *interact_json* lambda function because it is something specific to Ethereum RESTful API, anyway all the code is available on GitHub at https://github.com/Shotokhan/blockchain-benchmarking-on-mininet. The amount of data sent with a transaction is almost the same if you change the amount of tokens you sent to an address, so we didn’t record it as an environment parameter; furthermore, using different fees can change the order in which transactions are processed, but doesn’t impact the TPS rate at all. That’s all for the load-generating client. The observing clients are responsible for the measures of TPS. In fact, CPU and RAM usage can be measured by any process, because there isn’t any process virtualization, and traffic is measured just on one host: we make the hypothesis that the volume of traffic measured on an host is a good estimation of the volume of traffic perceived by other hosts. That said, the algorithm of the observing client is the following: it makes a read operation on the blockchain to fetch the height of the current head of the blockchain, then it enters in a *polling* state, waiting for the height to increase; as long as the height increases, and note that it can increase of more than just one in a single step, for reasons of unavailability, the observer goes through all new blocks, adding the number of transactions that is in each block to a counter, until the actual height of the blockchain is reached. It stops automatically after 200 seconds. It is implemented like a thread that executes the described algorithm in its *run* method and provides a read method which returns the TPS measure as the count of the transactions on the observation time. Note that, since this client counts transactions as long as new blocks are added to the blockchain and never goes back, its behaviour is AP. Below is an adapted and simplified code snippet, without the complexity of the exception handling, taken from the NXT observing client:

```
base_cmd = "curl -m 2 http://{}:6876/nxt?requestType=".format(hostToIp(self.observedHost))
get_chainStatus = "{}getBlockchainStatus".format(base_cmd)
post_cmd = base_cmd[:10]+"--data
'includeTransactions=False&height={}'+base_cmd[10:]+"getBlock"
blockchainStatus = self.observedNet.nameToNode[self.observingHost].cmd(get_chainStatus)
parsedBlockchainData = json.loads(blockchainStatus)
self.currentHeight = parsedBlockchainData['numberOfBlocks']
phasedHeight = self.currentHeight
start = time.time()
elapsed = time.time() - start
while elapsed < self.observingTime:
    while self.currentHeight == phasedHeight:
        time.sleep(1)
        blockchainStatus =
self.observedNet.nameToNode[self.observingHost].cmd(get_chainStatus)
        parsedBlockchainData = json.loads(blockchainStatus)
        phasedHeight = parsedBlockchainData['numberOfBlocks']
        elapsed = time.time() - start
        if elapsed >= self.observingTime:
            break
    while self.currentHeight < phasedHeight:
        blockData =
grep_json(self.observedNet.nameToNode[self.observingHost].cmd(post_cmd.format(self.current
Height)))
        parsedBlockData = json.loads(blockData)
        self.TX_count += parsedBlockData['numberOfTransactions']
        self.currentHeight += 1
        elapsed = time.time() - start
        if elapsed >= self.observingTime:
            break
    elapsed = time.time() - start
```

Note that, in both clients, we sent to hosts commands that are curl requests: the load-generating client and the observing clients make at all effects HTTP requests to blockchain nodes. Other implementation aspects are the definition of the topologies using the Mininet API, the automatic configuration of blockchain nodes before the bootstrap, code related to traffic capture, process monitoring, tests execution and automated analysis of the results to produce graphics and spreadsheets. The only thing of these that is interesting to record as environment parameter is the configuration of the overlay network, i.e. how peers discover each other. We configured it statically and full mesh; eventually, we verified from nodes logs that they don't get connected in a full mesh overlay network: they try their best to connect between them, for example if there are 8 peers, from the log of a node you can see that sometimes its peer count is 3, sometimes 6, sometimes the maximum that is 7, and sometimes even 0. This configuration part and all the rest of the code can be found on the GitHub repository highlighted in the link previously shown.

2.2.2 NXT

NXT's private evaluation kit was used for the experiments[31]. There are 10 hard-coded genesis recipient accounts with passwords “0” to “9” and 100.000.000 NXT initial balance each. It is configured in such a way to run only as a private testnet. You can run a node then access the client interface on localhost, port 6876. This is the same interface used by the load-generating client and the observing clients: instead of studying the API, we made some requests from the browser, captured data sent and received, and just replayed the same requests with some placeholder for input parameters like the passphrase or the block height. Therefore, we identified five REST HTTP calls of interest: *getBlockchainStatus* which is GET, *startForging*, *stopForging*, *sendMoney*, *getBlock*, which are POST. We used these calls to implement NXT specific tests; the responses were in JSON format.

Let's see the results of the first test case. The maximum amount of nodes which can run on the VM is 40, that is the maximum of the test case; going more than this would result in swap-in / swap-out operations from main memory to disk, as can be seen in the graphics.

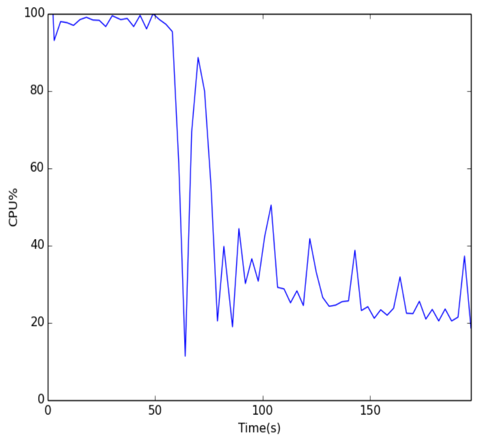


Figure 12: CPU over time, 40 peers in star topology NXT

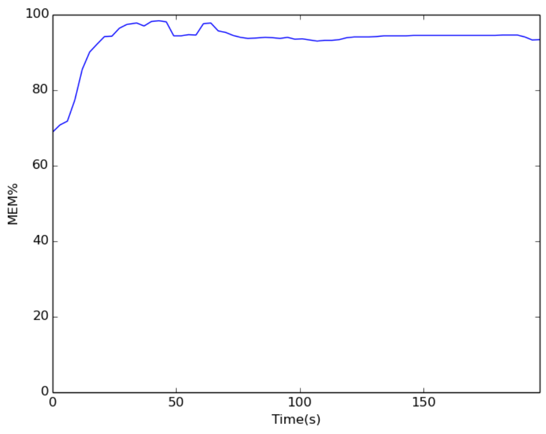


Figure 13: Main memory over time, 40 peers in star topology NXT

Because there aren't tip transactions and there is a lot of competing traffic on a star topology with 40 nodes, in this experiment the traffic exchanged between pairs of nodes was low and they didn't succeed in forging more than one block. That's because a star topology doesn't divide broadcast domains, therefore ARP requests produce a lot of noise. The tree topology provides better scalability and a very good separation of broadcast domains, so we evaluated insights about traffic only from the experiments that involve the tree topology. Now, we go on with the results of the second test case. We start with the parametric plots, then we explore a case we choose by looking at the plots.

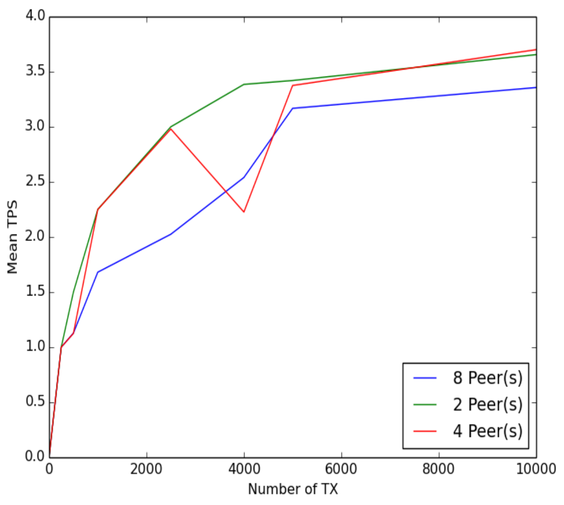


Figure 14: NXT tree topology parametric plots, TPS

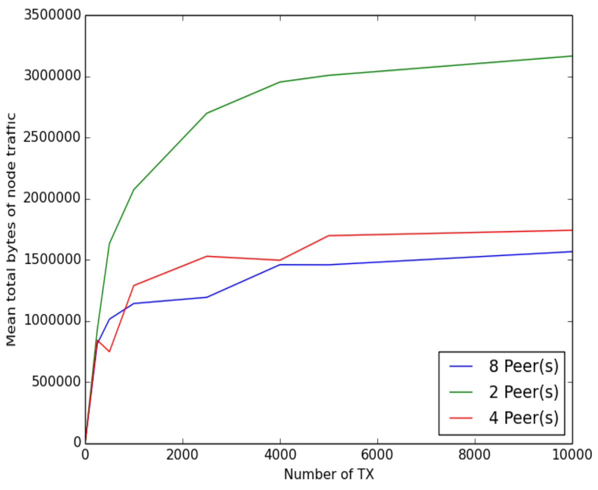


Figure 15: NXT tree topology parametric plots, traffic volume

Remember that a number of transactions in input equal to 5000 means an offered TPS rate of 25, in our experiments, so for example 10000 is 50 TPS. Note that, as the number of transactions increase, the mean TPS in output increases slowly. It is "mean TPS" because it is the mean of the TPS rates measured on each blockchain node: remember that observing clients are AP, so they can measure different TPS rates. Under 1000 transactions, i.e. under a load of 5 TPS, the behaviour is similar with respect to the number of peers. For what we said about the blockchain work metrics, this means that 8 peers are making three times the work of the 2 peers, providing the same TPS and at the same time providing more security and availability. After a certain point, the curve related to the topology with 8 peers keeps under the other two curves. Despite that, the blockchain work done by 8 peers is higher than the work done by the other two topologies. In fact, TPS rates for tree depth equal to 1, 2 and 3 are respectively equal to 3.66, 3.7 and 3.36: then blockchain works are respectively equal to 3.66, 7.4 and 10.08. It could look strange that the traffic measured on the interface of a peer, i.e. the bandwidth used by a peer, is higher with less peers. The motivation of this is the fact that the traffic measured on a peer takes into account also read operations and transactions. What actually boosts the curve of the traffic is the traffic generated by transactions: since transactions are sent in a round-robin fashion, if there are less peers then there is less load balancing, i.e. a single peer has to process more transactions. For example, a burst of 50 transactions every 2 seconds against 2 peers means 25 transactions every 2 seconds sent to each peer, that is 100 transactions every 8 seconds. If there are 8 peers, then each one of them receives 25 transactions every 8 seconds, and so on. Furthermore, it appears that with NXT protocol, nodes compete in forging blocks to maximize their return from transactions fees, therefore not all transactions that a node receives are bounced in broadcast to other nodes: they don't try to create a "shared transaction pool". This means that there is less collaboration between peers and more competition.

That said, we will explore deeper the case of 8 peers, with 10000 input transactions.

Left peer	Right peer	Right to left frames	Left to right frames	Total frames	Duration
10.0.0.8	10.0.0.1	99	239	338	200.0108
10.0.0.7	10.0.0.1	115	97	212	199.9789
10.0.0.2	10.0.0.1	95	93	188	199.3403
10.0.0.6	10.0.0.1	78	76	154	201.1532
10.0.0.3	10.0.0.1	50	100	150	201.4607
10.0.0.4	10.0.0.1	74	75	149	190.3376
10.0.0.5	10.0.0.1	49	77	126	194.9155

Table 3: Traffic between peers as seen by h1, NXT

Remember that we capture traffic always only on the host h1, for the hypothesis done in the Implementation of tests' paragraph. We removed relative start and bytes' information from these table (they were included in tshark output), because we don't care here about that. We just have to note the fact that there isn't the same amount of traffic exchanged with all peers, despite they are all connected for a similar duration. This can be interpreted as follows: from the point of view of h1, h7 and h8 are more active in the consensus mechanism with respect to others. This doesn't mean that other peers are less active in the global view: for example, h3 could be active towards h4, and so on.

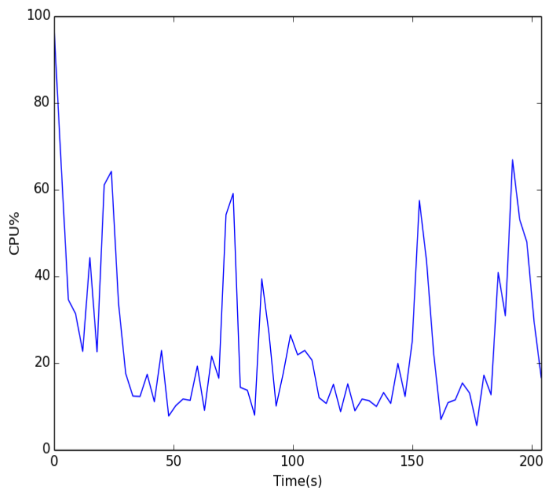


Figure 16: CPU over time, tree topology of depth 3 with 10000 transactions NXT

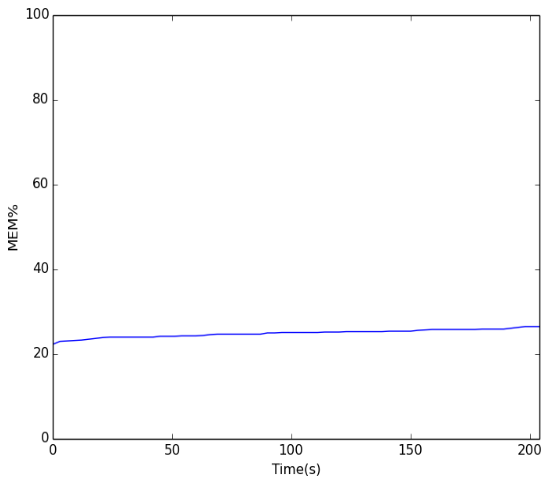


Figure 17: Main memory over time, tree topology of depth 3 with 10000 transactions NXT

As you can see, there is still a lot of free main memory: in this configurations, there are 17 hosts, because 8 are blockchain nodes in execution, 8 are observing clients and the other one is the load-generating client, and we showed that we can have up to 40 nodes running without swap operations; clients that are not nodes require very little main memory because they are just shell processes. The CPU usage over time has very high spikes due to the forging algorithm. The third test case is about the link going up and down. As you can expect, in this situation the result is very different.

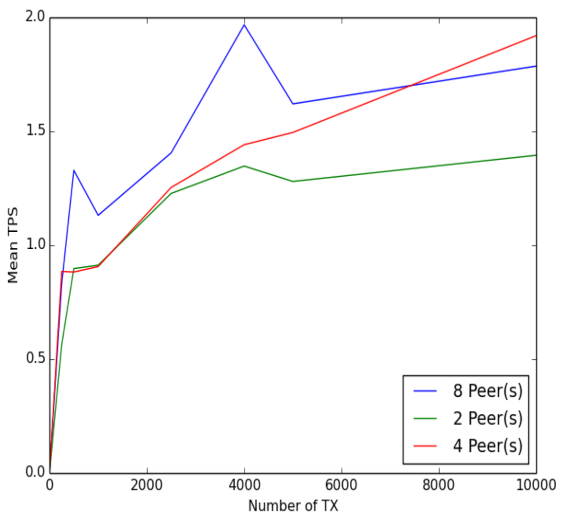


Figure 18: NXT tree topology with link up and down parametric plots, TPS

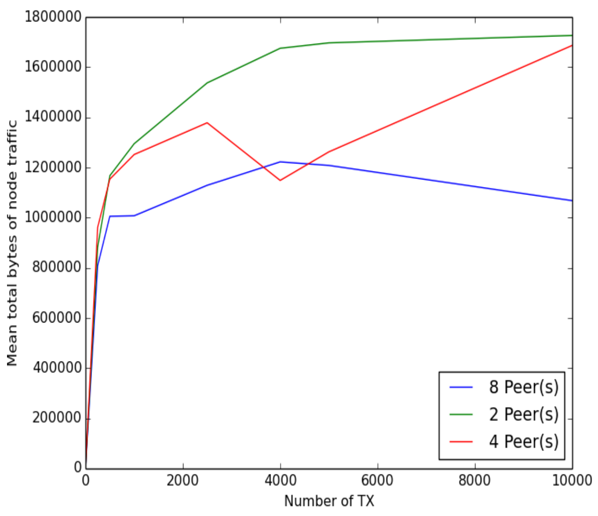


Figure 19: NXT tree topology with link up and down parametric plots, traffic volume

The transaction throughput is less for any number of peers: this is the result of the network partition. In the case of 2 peers, they forge lonely for an amount of time: the fact that they remain isolated is equal to a total network failure, therefore the low throughput is natural. The amount of traffic is also less because they just can't connect among them for a fixed period of time. We verified from nodes' logs that eventually, peers reached consensus again; you can see something like that:

```
...
2020-08-27 20:28:32 FINE: Accepted new transaction 15277233537795098106
2020-08-27 20:28:35 FINE: Will process a fork of 6 blocks, mine is 5
2020-08-27 20:28:35 FINE: Rollback from block 18017916947786391919 at height 168 to
7866310397944065253 at 163
2020-08-27 20:28:37 FINE: Switched to peer's fork
2020-08-27 20:28:37 FINE: Forger 11362740741557265300 deadline 56 hit 109750113
2020-08-27 20:28:37 FINE: Accepted new transaction 6745837038950370063
...
2020-08-27 20:29:04 FINE: Account 11362740741557265300 generated block
81946891628790942 at height 172 timestamp 109750125 fee 626.0
2020-08-27 20:29:04 FINE: Accepted new transaction 17234040967001055141
...
```

The moment in which the consensus on the valid chain was reached was highlighted in red: the node which was logging this file reported that it switched to another fork. You can also note that it first rolled back from an height of 168 to an height of 163, then after 30 seconds it was at an height of 172 (in the testnet configuration, nodes forge a block approximately every 15 seconds, so it is clear that it had to switch to a longer fork to reach that height so fast).

2.2.3 Ethereum

We used geth, the official Ethereum client implementation in Go. To use it on a private testnet, you have to first create an Ethereum account for each node. Then, you use an utility called Puppeth to create a genesis block. The only problem is that Puppeth is interactive and you have to specify the addresses of the accounts; since in our experiments we create new accounts for every single test case, we created a genesis block once, and then modified it for any new experiment. It was easy because it comes in a JSON format. An environment parameter here is the set of nodes allowed to sign blocks, since we use Ethereum's Clique protocol, which is proof of authority. The automatic configuration algorithm always sets h1 and h2 as nodes allowed to sign: there are two signers. For how Clique protocol works, they have to alternate in signing blocks. Therefore, even with more peers, they're always the group leaders. This time we didn't have a cool client interface to try and intercept HTTP requests, so we read the API documentation[33]. The requests were all POST, and they accepted JSON as input and provided JSON as output, then the tests' code resulted cleaner. We used the following methods: *eth_blockNumber*, *eth_getBlockByNumber*, *eth_sendTransaction*. From the documentation: "An uncle doesn't contain individual transactions". Then, we're fine with these methods, i.e. our observing client does the right thing, because transactions belonging to uncles are all included in the block of the canonical chain, to make read operations on the blockchain easier, without the need to check for uncles to look for additional transactions.

Let's see the experimental results for Ethereum.

Before swap operations, just as few as 10 nodes can run at the same time on the VM:

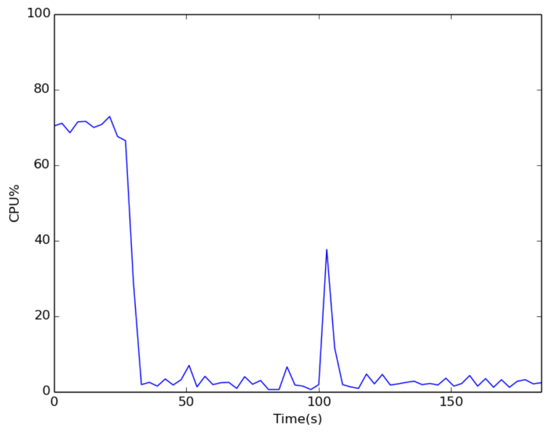


Figure 20: CPU over time, 10 peers in star topology Ethereum

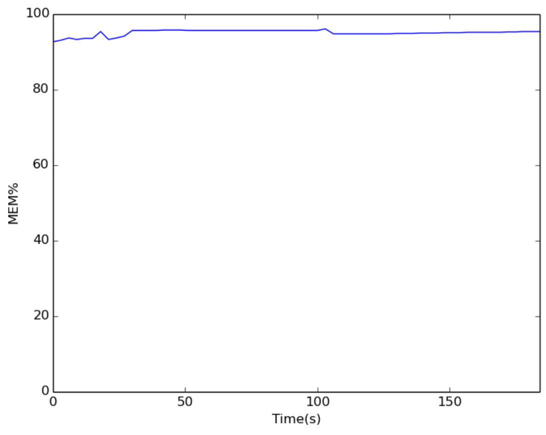


Figure 21: Main memory over time, 10 peers in star topology Ethereum

This penalizes CPU usage: if you want a good CPU usage, you need more memory or you need to rely on swap operations. This is not justified, as we're using Clique protocol, and not Dagger. About the traffic, the same considerations done about NXT apply: forging isn't successful in star topologies. Let's go on with the second test case.

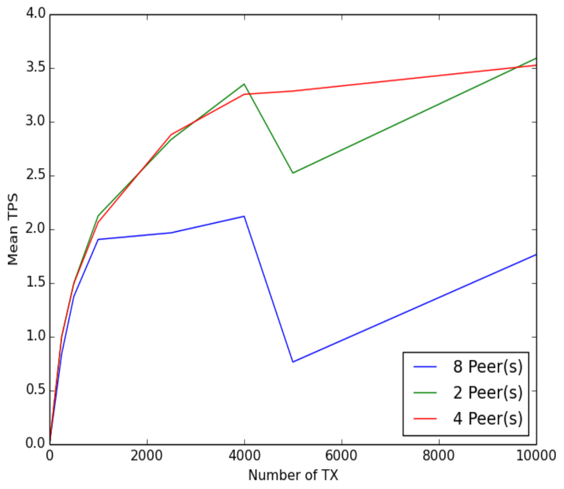


Figure 22: Ethereum, tree topology parametric plots, TPS

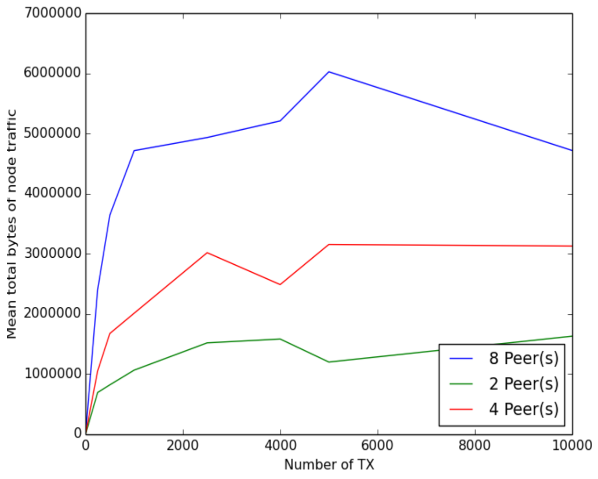


Figure 23: Ethereum, tree topology parametric plots, traffic volume

As the number of transactions approaches 10000, the blockchain work done by 8 peers approaches 4.5, that is more than the work by 2 peers and less than the work done by 4 peers that is 7, despite the traffic generated on a peer's interface gets higher as the number of peers increase. This is the result of the proof of authority protocol: although the load balancing of transactions, nodes still have to send transactions to authorized signers in order to make them add transactions to “sealed blocks”. This is the evidence of an higher degree of collaboration. In our experiments, there always were only two authorized signers; to implement a form of load balancing in a proof of authority scheme in which only one signature is required to confirm a block, more signers are required. Having only two signers was fine in the case of up to 4 peers, but more are required in the case of more peers, otherwise the little set of signers is overwhelmed with frames, as shown in the following table (8 peers, 10000 transactions).

Left peer	Right peer	Total frames	Total bytes	Duration
10.0.0.5	10.0.0.1	8411	722313	220.9279
10.0.0.3	10.0.0.1	8048	680255	221.0028
10.0.0.2	10.0.0.1	8020	686735	221.6732
10.0.0.6	10.0.0.1	7976	683467	221.5175
10.0.0.8	10.0.0.1	7919	685579	220.9946
10.0.0.7	10.0.0.1	7414	629949	221.1932
10.0.0.4	10.0.0.1	3845	331612	196.5496

Table 4: Traffic between peers as seen by h1, Ethereum

This time we have shown total frames and total bytes, because right to left and left to right amounts are very similar. As you can see, a lot of frames were exchanged to implement the consensus mechanism.

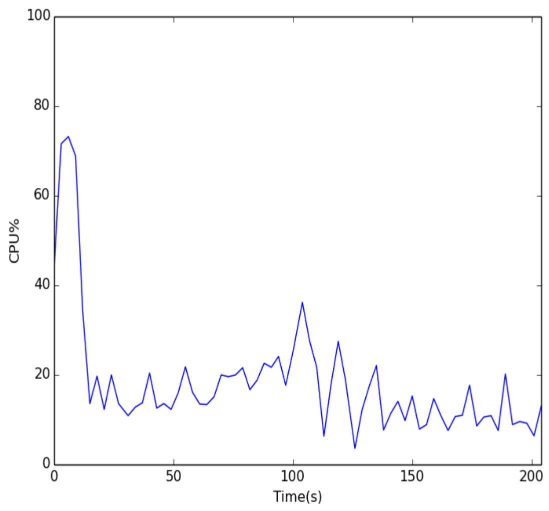


Figure 24: CPU over time, tree topology 3 depth with 10000 transactions Ethereum

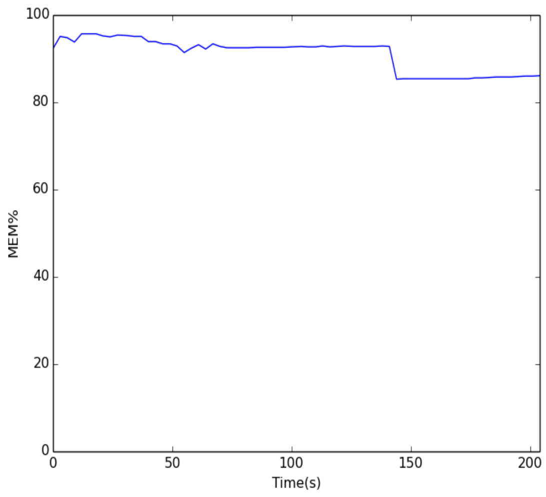


Figure 25: Main memory over time, tree topology 3 depth with 10000 transactions Ethereum

There aren't very high CPU spikes because there is no forging algorithm in action, only signatures. However, we're close to swap operations: main memory usage is very high.

At this point you may wonder how would a proof of authority protocol react to a network partition, that is done in such a way to cut the communication between non-signer nodes and signer nodes. Parametric plots showing TPS and traffic are similar in shapes to these of figures 22 and 23.

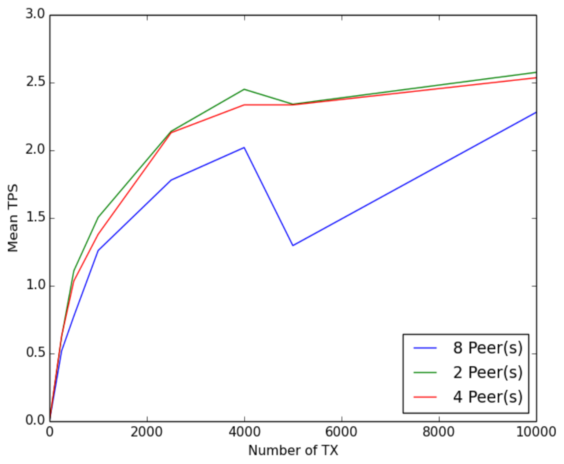


Figure 26: Ethereum, tree topology with link up and down parametric plots, TPS

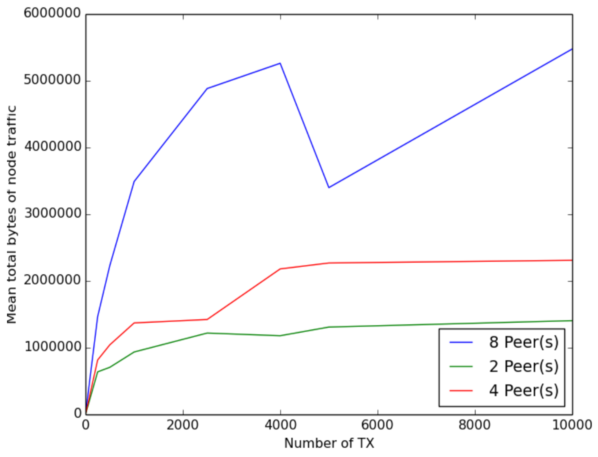


Figure 27: Ethereum, tree topology with link up and down parametric plots, traffic volume

The interesting thing is that nodes accumulate transactions, and then they send to authorized signers as soon as they are able to: the topology with 8 peers performed even better with the link down. This discussion applies also to the case of 2 peers that are both signers, which are forced to wait for each other. Let's see this thing from nodes' logs, during the transition in which the link switches from down to up, in the case of 2 peers with 10000 transactions.

```
...
INFO [08-27|21:17:07.815] Commit new mining work                number=3
sealhash="45950c…05ec8f" uncles=2 txs=223 gas=4683000 fees=4.683e-12
elapsed=17.451ms
INFO [08-27|21:17:07.815] Signed recently, must wait for others
INFO [08-27|21:17:08.208] Submitted transaction
fullhash=0x511fe2ebb6fbe056eb9bde21038aae71fee4078141248a9f6833b4f133a471e7
recipient=0xFb9a175032Adbd79e54258cD1B4CE87F8B17E8aa
INFO [08-27|21:17:09.686] Submitted transaction
fullhash=0x81e7d0e602c9c770750562e63f2fcee5fc64a6399dee0b6fd6b35d5f7b2cb3a1
recipient=0xFb9a175032Adbd79e54258cD1B4CE87F8B17E8aa
INFO [08-27|21:17:10.106] Submitted transaction
fullhash=0xfe9c0cf2972612a90121fa52d2f44e4e28e2decac9311bb662d2438a0639f52c
recipient=0xFb9a175032Adbd79e54258cD1B4CE87F8B17E8aa
INFO [08-27|21:17:10.237] Chain reorg detected              number=1
hash="1f5aa4…99adca" drop=1 dropfrom="ff5a3d…95e2bc" add=2
addfrom="217374…46b2ed"
INFO [08-27|21:17:10.239] Imported new chain segment            blocks=1 txs=147
mgas=3.087 elapsed=115.536ms   mgasps=26.719 number=3 hash="217374…46b2ed"
dirty=5.28KiB
INFO [08-27|21:17:10.239] Commit new mining work                number=4
sealhash="bb68f9…70651f" uncles=2 txs=0   gas=0      fees=0
elapsed="700.798µs"
INFO [08-27|21:17:10.247] Commit new mining work                number=4
sealhash="5d43cc…fae8f8" uncles=2 txs=6   gas=126000  fees=1.26e-13
elapsed=8.131ms
INFO [08-27|21:17:10.589] Submitted transaction
fullhash=0x58c85f2a773617c87ff9176e284b40cd7f4a568645d23a0a109523d1788dfd06
recipient=0xFb9a175032Adbd79e54258cD1B4CE87F8B17E8aa
...
```

The line highlighted in red is the one where the logging node imports the chain segment from the other node. This doesn't happen only in cases of links going up and down: this is an usual operation due to the "accumulation of transactions" in nodes who can't sign blocks. You can also see in action the fact that a signer can only sign one block out of K blocks, and after that it must wait for others: "Signed recently, must wait for others". The careful reader will also note the “uncles” field in the “Commit new mining work” lines: this is the GHOST protocol in action.

2.2.4 Comparison of the results

We analysed part of the results for both NXT and Ethereum in the last two sub-paragraphs. NXT nodes’ software proved to be more scalable: although it uses more CPU power to execute a forging algorithm, it requires less main memory and less bandwidth for each peer. The fact that less bandwidth is required for each peer also means that the total amount of traffic generated on the underlying network infrastructure is less. We recall that the reason behind the bandwidth requirement for Ethereum is Clique protocol, that is proof of authority. But there are positive aspects: there is more collaboration between peers in Ethereum and thanks to the GHOST protocol, the amount of traffic wasted is less. In contrast with that, a significant part of the traffic generated by NXT is wasted: when a fork becomes the valid chain, the blocks of the other fork which were not included in the valid chain become garbage, and if they were previously sent in broadcast to other peers, then all this garbage is traffic wasted. These reorganizations of the chain happen as a result of the nature of the proof of stake consensus mechanism. In terms of TPS and consequently in terms of blockchain work, NXT performed a little bit better in the second test case, with the performance gap that gets more evident as the number of peers increase: again, PoS provides more scalability than PoA if there aren’t enough authorized signers who share the task of confirming blocks. However, in the third test case, i.e. with a network partition, Ethereum outperformed NXT: maximum TPS rates for Ethereum in this test case reach and go beyond 2.5, whereas NXT is always under a rate of 2 TPS. This is thanks to the combined action of PoA and the GHOST protocol, which don’t make wasted the efforts of the two separate partitions, whereas in the PoS scheme with cumulative difficulty, a fork will be selected, the other will be discarded and with it all the transactions that were confirmed, because nodes don’t share their transaction pools, at least not at all.

2.3 Comparison of the test setup with Hyperledger Caliper’s one

According to [34], Caliper is a general framework for executing benchmarks against different blockchain technologies. It is modular, scalable and extensible: it claims to be easy to integrate with popular monitoring and infrastructure solutions. It is a service that generates a workload against a specific SUT , monitors its responses and generates a report from them. An high-level depiction of Caliper architecture is the following:

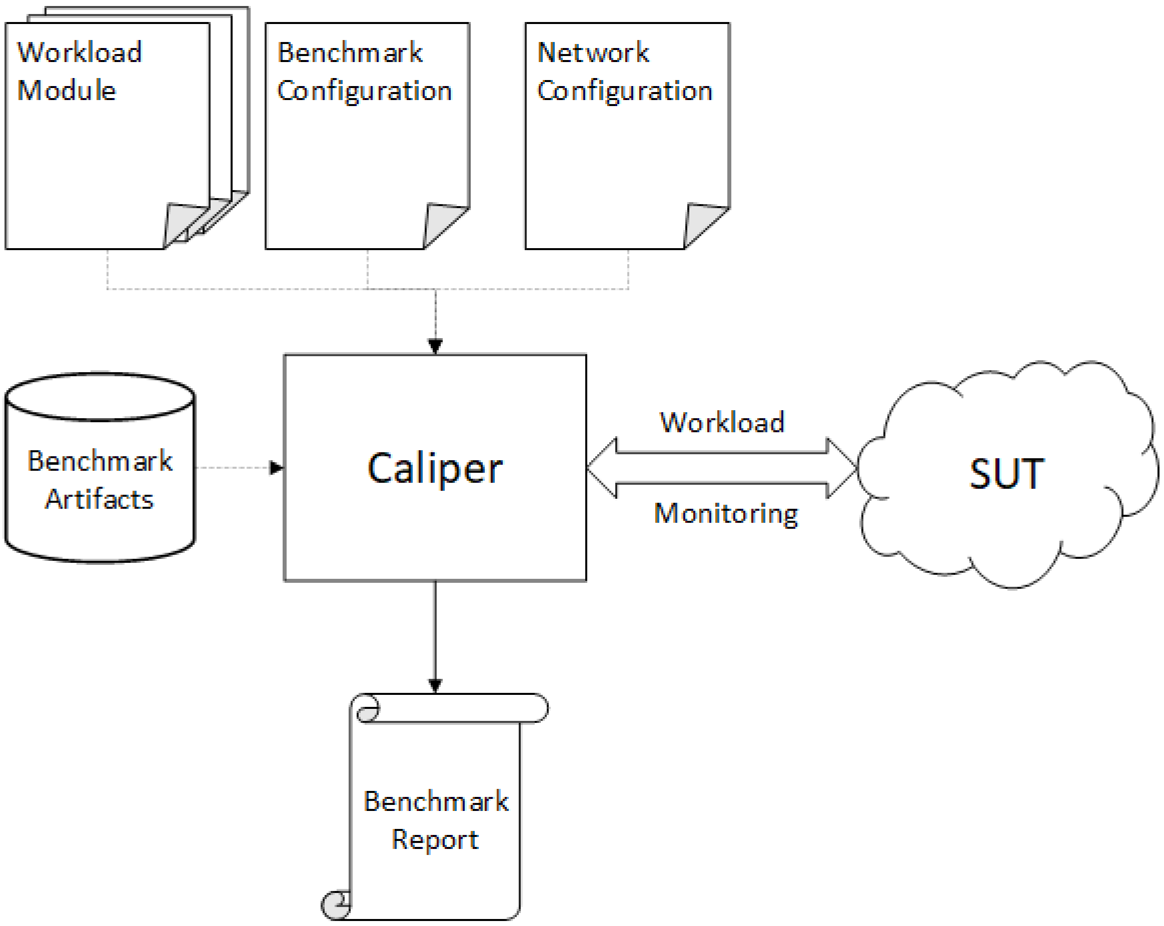


Figure 28: Caliper's high level architecture (Source: [34])

As can be seen in the figure, Caliper requires several inputs to run a benchmark.

The benchmark configuration file is the “flow orchestrator” of the benchmark: most of the settings are independent of the SUT and therefore they can be used to perform multiple benchmarks against different SUT types. These settings include the number of rounds that should be executed, the TPS rate in input, which workload module will generate the content of the transactions, parameters about monitoring.

The network configuration file is SUT-specific: it describes the topology of the SUT, the addresses of the nodes, the smart contracts that should be deployed, if any.

Workload modules are the key for the execution of a benchmark: Caliper is a general benchmarking framework, therefore it doesn’t include any concrete benchmark implementation. When Caliper schedules transactions for a given round, it is the task of the round’s workload module to generate the content of the transactions and submit it. In other words, workload modules must export a given API. They are Node.JS modules: everything you can code in Node.JS can be the logic of a workload module.

Benchmark artifacts are additional artifacts that may be needed to run a benchmark which varies between different benchmarks and runs, for example, pre-installed third-party packages for workload modules.

Another peculiarity of Caliper is the multi-platform support: taking the best from adapter and facade design patterns, Caliper provides an interface to make operations like “initialize the SUT”, hiding the details of different SUT types: the adapter implementation takes care of the rest. Note that the implementation is specific to the SUT, of course, but it must always provide the same interface.

One of the most important goals of Caliper is scalability. If you generate all the workload on a single machine, you can quickly reach the resource limitations of that machine. To make sure that the workload rate matches the scalability and performance characteristics of the evaluated SUT, a distributed approach is needed. The implementation of this comes with the manager/worker model. The manager initializes the SUT, coordinates the run of the benchmark, and handles the performance report generation based on the observed statistics about transactions. The worker processes perform the workload generation, independently of each other. Worker processes are the key point behind Caliper’s scalability, because they can be deployed on different machines. There are three process distribution models:

- Automatically spawned processes on the same host, using inter-process communication with the master process.
- Automatically spawned processes on the same host, using a remote messaging mechanism for the communication with the master process.
- Manually spawned processes on multiple hosts, using a remote messaging

mechanism for the communication with the master process.

Of course, the third method is the one which provides scalability. Note that we didn't say yet where the SUT is deployed: that's because, thanks to the SUT adapter, it can be deployed everywhere. With this observation, we can look at the architecture of the processes:

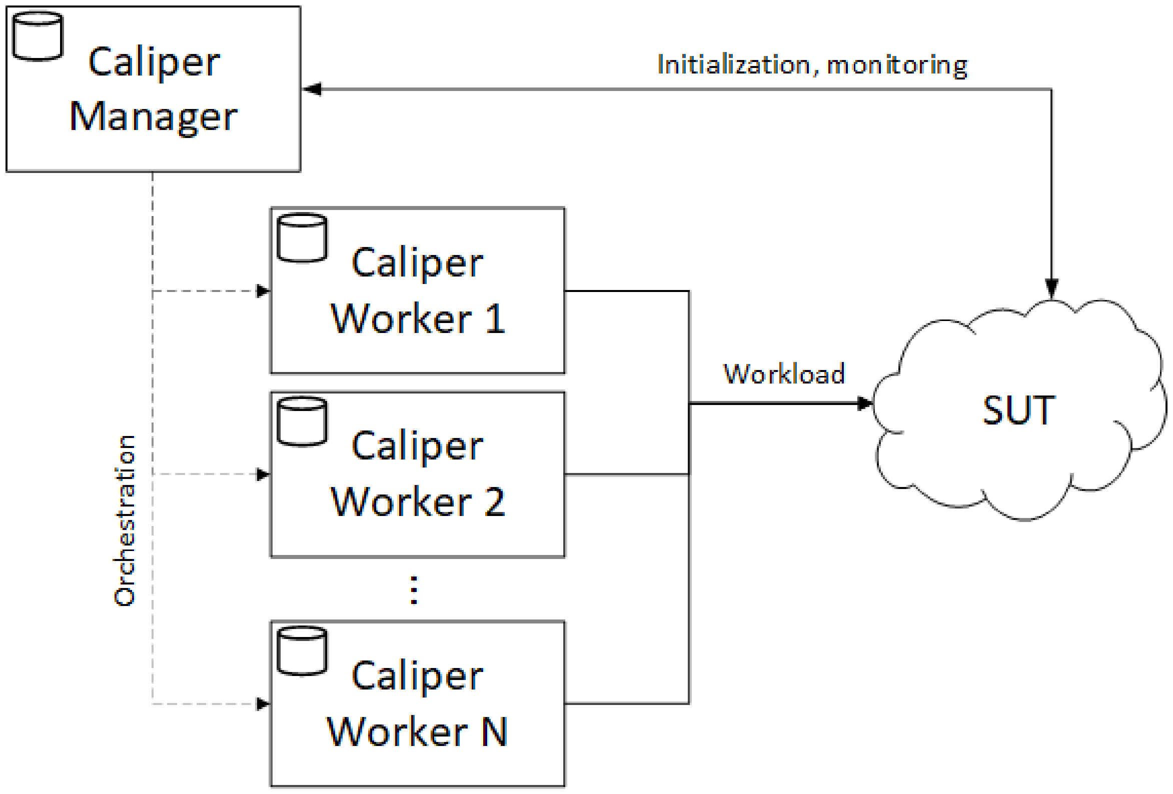


Figure 29: Caliper's process-level architecture (Source: [34])

To achieve a great workload rate, the best thing is perhaps a fully distributed deployment. There are Caliper Docker images to make the integration with solutions for automatic deployment and management painless.

At this point, we can compare our setup with Caliper's one. The first thing that comes in mind is that Caliper uses configuration files to define benchmark settings such as time of the experiment, TPS rate and so on, and also to define network settings. In our experiments, these settings are cabled into the code. Caliper's approach is not only more modular, but also leads to a self-documented benchmark: we needed to record all environment parameters manually, whereas Caliper configuration files are both the settings and the documentation. In addition to that, doing so avoids the repetition of some parameters in many modules: workload modules are only responsible for the SUT-specific

API, whereas our test case modules also contain the benchmark settings. Our network settings are well separated from the test cases because we used Mininet API, but the approach is not flexible to SUTs that are not built upon Mininet. Another aspect is the SUT adapter: whereas Caliper divides the modules for the injection of workloads and for the bootstrap of the SUT, providing a standard interface to start/stop the SUT, we made our test cases responsible for starting and stopping the SUT. Furthermore, our approach is not scalable, because it is only intended to run on Mininet, which at the time of writing only runs locally: therefore, making an analogy, the SUT, the manager and all the workers run on the same host. At last, Caliper seems easily deployable on any system thanks to Docker images, whereas our setup is easily deployable on Mininet VM but requires many dependencies to be installed on other Linux distributions and is not deployable on other operating systems. A side in which our setup is similar to Caliper's one is the network side: in our emulated network, observing clients and the load-generating client are separated from the hosts which execute blockchain nodes, and this fact leads to a traffic generated similar to the traffic that would be generated by Caliper's setup, except from the fact that in our case, the only mean of communication between the "manager" and the "workers" is inter-process communication, that is to say the pipes generated by Mininet. Another common point is the reporting: our reporting is enough standard and the modules for the analysis are well separated from the other modules.

Conclusions

We made a general survey about blockchain technologies, brushing different examples and diving deeper into two technologies, NXT and Ethereum. We analysed these two technologies both from a theoretical and an experimental point of view. The experimental setup took the most of a scalable emulated network environment: Mininet. The test suite defined was enough to make an interesting comparison between NXT and Ethereum, from different points of view: performance, scalability, partition tolerance. Lastly, we compared our setup with the better engineered setup developed by Hyperledger, that is a project initiated by the Linux Foundation which formed a group of industry leaders like Accenture, Cisco, IBM, Intel and even banking institutions[35]. We conclude with some considerations about possible improvements of our setup and with a consideration about putting together blockchain technologies with other disruptive technologies.

The first improvement we could do is about the probability distribution of input transactions in workloads. In our setup, we have a fixed number of transactions in a single burst, and, provided the total number of transactions, a fixed amount of time between two consecutive bursts. If we design the number of transactions in a burst, let's call it burst range, and the amount of time between two consecutive bursts, let's call it delay, as random variables, then we have uniform random variables. In real life, in most cases you don't have uniform random variables. Moreover, transactions are not forced to come up in bursts. We could then make bursts of 1 and bring the scheduling of nodes apart, so that after the delay the node is chosen. Nodes can still be chosen in a round-robin fashion, but this is another form of uniform random variable. So, having eliminated the burst, we still have two random variables: one for the delay and one for the selection of the node to which sends a transaction. An option could be to model the delay as a Poisson variable, using the parameter expected value λ in such a way to adjust the offered TPS rate in input:

$$P(x=K)=\frac{\lambda^k e^{-\lambda}}{k!} \quad \text{(C.1)}$$

In practice, you can run a simulation, with k elapsed time since last transaction, and you discretize the time: at each time step, you “throw a coin” with the probability expressed by formula C.1, and if the coin flips right, you send a transaction. But to send a transaction, you have to choose to which node your transaction must be sent. To make this choice, any distribution can be chosen: you build a population of nodes, and you assign a weight to each node. The higher is the weight, the higher is the probability of the node of being chosen. Note that the weights can also be random variables themselves.

The second improvement is about a better engineering of the setup, taking something from Caliper’s architecture. For example, the parameters of workloads in configuration files is a good idea. But before that, we should first re-organize the project in order to achieve high cohesion and low coupling for all modules. At the time of writing, the modules are pretty much low coupled, but in the test modules there is low-intermediate cohesion: these modules do too much. Making an analogy with Caliper, our test modules are the manager, the workers, the SUT and even call the functions to make the analysis. It’s okay to have a main module that does these things, but it needs a finer level of granularity.

Other features that would be interesting to add are the “faultloads” and a CP observing client. An example of faultload is a double-spending attack: we know that, in theory, NXT and Ethereum’s consensus mechanisms protect against this attack, but under the hypothesis that clients don’t trust blocks that are on the head of the chain. What is the impact of an aggressive double-spending attacker who tries to perform the attack many times? It would be an interesting paragraph. About the observing client, from our experiments we obtained as result that, in almost every single test case, some observing clients measure TPS rates different from others. This is more likely to happen when there is the network partition and is more a problem of NXT than of Ethereum, again because Ethereum uses PoA in combination with GHOST. Here is an example of that, in the case of NXT, tree with 8 peers, 10000 transactions and link up and down:

Observer node	Measured TPS
h8	1.265
h2	2.28
h3	2.49
h1	1.03
h6	1.265
h7	2.385
h4	2.49
h5	1.085

Table 5: TPS rates measured by peers

The motivation behind this result is the nature of the AP observing client. It reads the blockchain height, compares with the height it reached, and when the blockchain is above it goes on counting transactions, never going back. It wouldn’t know whether another fork was selected or not: it just measures transactions confirmed in real-time on the peers, not caring about the consistency. This leads to an approximation of the real throughput offered by the blockchain network. To solve this issue, a CP observing client is required. To implement it, the client should read the blockchain height when it starts, make a polling like the one done by the AP client but without going on with its local height, and at the end of the experiment time it should run across all blocks from the initial to the last heights read, counting transactions. It still could lead to bad results: to make sure that blocks are consistent in the overlay network, the SUT must keep executing in order to forge another X blocks, for example 7 blocks, and the CP client has to wait for that. Furthermore, the SUT must keep executing also after the forging of these additional blocks, for the time required to the CP client to make the read requests needed for the measure. Note that we spoke of CP client and not CP clients: there should be only one valid chain segment across all peers in the height interval that has to be queried by the client, otherwise we couldn’t speak of “consistency”.

The last thing we want to highlight is the potential of blockchain technologies of matching well other disruptive technologies, such as big data analysis and cloud technologies[1]. An useful example to go straight to the point is Xbox’s private blockchain. The company used a blockchain for smart contracts which involve game publishers and musicians: these smart contracts aim at simplifying the management of the so called royalty payments. When a game publisher includes music in a game, it has to pay the related musicians for the use of the intellectual property. These are the royalty payments. There can be contracts like: “if the game earns more than X, then pay Y % of the earning to the musician in addition to the base payment of B”. Before blockchain, managing these payments was an hard paper work, and it could take several weeks for a game publisher to get on-boarded, due to the inefficiency of the system. With blockchain, royalty payments were managed with smart contracts: the on-boarding time went from several weeks to 15 minutes. Xbox didn’t build a new blockchain technology itself from scratch: it used Azure’s *blockchain-as-a-service*. Even with little knowledge about blockchain technologies, you can implement your logic on the blockchain using this service, taking the most of Azure’s cloud security, reliability and storage solutions, not to mention the blockchain code itself. This is the first match: blockchain with cloud technologies. About the second match, think at the fact that, before blockchain, Xbox’s royalty payments were managed in something like spreadsheets, with different structures, therefore trying to analyse that data to gain some insights was nearly impossible. With blockchain, all the data is forced to have the same structured format. Therefore, it matches well with big data analysis and machine learning tools: if a game publisher wants to monitor how its music costs have grown over time, it can do it in a few minutes, maybe with something like a dashboard for data analysis. If you want to gain more insights about transactions and entities involved in transactions, it’s easy to apply machine learning techniques because data is well structured. This result is not new: “disruptive” technologies work better together, when they feed each other,

Bibliography

[1] Neel Mehta, Adi Agashe, Parth Detroja, “Bubble or Revolution? The Present and Future of Blockchain and Cryptocurrencies”, Paravane Ventures, 2020

[2] Satoshi Nakamoto, “Bitcoin: A Peer-to-Peer Electronic Cash System”, https://bitcoin.org/bitcoin.pdf

[3] Nxt community, “Nxt whitepaper”, Revision 4 – Nxt v1.2.2 – July 12, 2014

[4] Thomas H. Cormen [et al.], “Introduction to Algorithms” (3rd ed.), MIT Press, 2009

[5] SoluLab, answered on 23/12/2019: https://www.quora.com/What-kind-of-databases-are-used-by-Blockchain-platforms

[6] Coordicide Team, IOTA Foundation, “The Coordicide”, May 2019

[7] Dominique Guegan. Public Blockchain versus Private blockhain. 2017. ffhalshs-01524440f

[8] Jonathan Ore, “How a $64M hack changed the fate of Ethereum, Bitcoin's closest competitor”, CBC News, Aug 2018

[9] Ekin Tuna, Gardener Oracle, “An unexpected but ideal union: Magic and Blockchain”, Medium, Aug 2019

[10] Vitalik Buterin, “Ethereum: The Ultimate Smart Contract and Decentralized Application Platform”, Bitcoin Magazine, Dec 2013

[11] The Coin Rise, https://thecoinrise.com/how-do-bitcoin-transactions-work/, Aug 2020

[12] Wenbo Wang [et al.], “A Survey on Consensus Mechanisms and Mining Strategy Management in Blockchain Networks”, arXiv:1805.02707v4 [cs.CR], 19 Feb 2019

[13] Binance Academy, “What is Delegated Proof of Stake (D-PoS) – Explained For Beginners”, https://www.youtube.com/watch?v=OVKAOwzAwHI, Feb 2019

[14] REST Bitcoin API, https://rest.bitcoin.com/, Aug 2020

[15] Julian Browne, “Brewer’s CAP Theorem: The kool aid Amazon and Ebay have been drinking”, http://www.julianbrowne.com/, written in Jan 2009, last access in Aug 2020

[16] Wikipedia, https://en.wikipedia.org/wiki/CAP_theorem, Aug 2020

[17] Yaron Y. Goland, “The block chain and the CAP Theorem”, https://www.goland.org/, written in Mar 2017, last access in Aug 2020

[18] Nicola Atzei [et al.], “A survey of attacks on Ethereum smart contracts”, Università degli Studi di Cagliari, Mar 2017

[19] Steemit, https://steemit.com/blockchain/@reverseacid/the-scalability-trilemma, Aug 2020

[20] Lee, Timothy B., “Bitcoin has a huge scaling problem – Lightning could be the solution”, Ars Technica, Feb 2018

[21] Bitcointalk, https://bitcointalk.org/index.php?topic=303898.0, Aug 2020

[22] Github, https://github.com/ethereum/EIPs/issues/225, written in Mar 2017, last access in Aug 2020

[23] Cryptocompare, https://www.cryptocompare.com/coins/guides/what-is-the-ghost-protocol-for-ethereum/, written in Jul 2015, last access in Aug 2020

[24] Blocking, https://blocking.net/5392/analysis-of-ethereum-ghost-agreement/, Aug 2020

[25] Vitalik, http://web.archive.org/web/20131228111141/http://vitalik.ca/ethereum/dagger.html, Aug 2020

[26] Unhashed, https://unhashed.com/cryptocurrency-news/ethereum-sharding-update-expected-2020/, Aug 2020

[27] Hyperledger, https://www.hyperledger.org/learn/publications/blockchain-performance-metrics, Aug 2020

[28] Bob Lantz, Brandon Heller, and Nick McKeown, “A Network in a Laptop: Rapid

Prototyping for Software-Defined Networks. 9th ACM Workshop on Hot Topics in Networks", October 20-21, 2010, Monterey, CA

[29] University of South Carolina, "Network tools and protocols: Lab 3: Emulating WAN with NETEM I: Latency, Jitter", 14 Jun 2019

[30] University of South Carolina, "Network tools and protocols: Lab 3: Emulating WAN with NETEM II: Packet Loss, Duplication, Reordering and Corruption", 14 Jun 2019

[31] Jelurida, https://www.jelurida.com/nxt/evaluation, Aug 2020

[32] Github, https://github.com/ethereum/go-ethereum, Aug 2020

[33] Ethereum Wiki, https://eth.wiki/json-rpc/API, Aug 2020

[34] Hyperledger, https://hyperledger.github.io/caliper/, Aug 2020

[35] LinuxFoundation, https://web.archive.org/web/20170717193806/https://www.linuxfoundation.org/news-media/announcements/2015/12/linux-foundation-unites-industry-leaders-advance-blockchain, written Dec 2015, last access in Aug 2020