Vrije Universiteit Amsterdam

Universiteit van Amsterdam

Master Thesis

SubSonic: A Wait-Free Concurrent Execution Model of Blockchain Transactions

Author: Kian Paimani (2609260)

1st supervisor:	dr. ir. A.L. Varbanescu	
daily supervisor:	supervisor name	(company, if applicable)
2nd reader:	supervisor name	

A thesis submitted in fulfillment of the requirements for the Master of Science degree in Parallel and Distributed Computer Systems

August 31, 2020

"In the future, trusting an opaque institution, a middleman, merchant or intermediary with our interest, would be as archaic a concept, as reckoning on abacuses today"

– Dr. Gaving Wood

Prelude

This will be prelude. Some wise words about Web3, and how it will evolve from web2 and how web2 was designed in a peer to peer fashion but ended up in this mess that it is now.

Abstract

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut ac orci a nulla finibus ornare. Proin ultricies tellus a metus facilisis tristique. Vivamus vel lectus magna. Donec nibh sapien, pulvinar ut hendrerit nec, porta nec est. Nam rutrum aliquet egestas. Etiam nibh ex, ultrices vel arcu eu, vulputate venenatis enim. Vestibulum vel nisi quis libero finibus sollicitudin. Aenean ornare nibh id tincidunt tempus. Duis imperdiet, dui sed finibus tempor, lacus enim tempor sem, eget fermentum enim magna sit amet nunc. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos himenaeos. In malesuada ligula risus, sit amet efficitur quam accumsan sed. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia curae; Mauris non felis sed felis euismod efficitur interdum eget massa. Nullam eget accumsan arcu. Nunc suscipit volutpat ligula sed blandit.

Vivamus lobortis leo est, gravida consequat risus volutpat eget. Morbi quam mi, porta ut vulputate in, gravida ut ipsum. Ut ultricies ullamcorper molestie. Suspendisse vestibulum eros eget elit finibus volutpat non vitae risus. Pellentesque gravida, lacus sit amet egestas fringilla, diam erat laoreet augue, id feugiat orci lorem vitae augue. Sed augue augue, volutpat et fringilla quis, dictum vel velit. Nam et congue diam.

Integer rhoncus ipsum ante, non sollicitudin nisi sagittis vel. Duis rhoncus consequat ante at semper. Vivamus ex turpis, blandit in nisi eget, hendrerit scelerisque libero. Donec posuere dui libero, vitae ornare orci tempus sollicitudin. Mauris ut tortor vel orci malesuada blandit in a felis. Phasellus at congue mauris. Sed efficitur tellus at ligula euismod rutrum. Aliquam mattis scelerisque ultricies. Integer rutrum erat a tellus ultricies cursus. Integer gravida suscipit nulla, interdum lacinia elit dictum sed. Donec luctus nibh ac tortor rutrum rutrum. Aliquam vitae fermentum ante, ac maximus turpis. Lorem ipsum dolor sit amet, consectetur adipiscing elit. Aliquam vel neque id felis viverra pretium.

Nullam scelerisque, orci quis porttitor rutrum, orci magna mattis neque, ac vestibulum dolor erat eget mi. Curabitur porta, lectus ac faucibus pellentesque, ante diam pretium nisi, ut maximus mauris lorem tincidunt eros. Mauris scelerisque elit ut ante auctor, in lacinia mauris dictum. Nam nec mattis nulla, eget tincidunt leo. Ut bibendum at mi nec commodo. Suspendisse augue massa, pharetra id pellentesque at, finibus at odio. Vestibulum ac enim ut tellus fermentum feugiat nec id eros. Duis hendrerit sem et ornare hendrerit. Suspendisse at sem vel tellus aliquet tempor. Fusce accumsan elementum est, sed aliquet nibh facilisis at. Sed venenatis condimentum magna. Curabitur iaculis augue in arcu ullamcorper placerat. Suspendisse ac placerat justo.

Maecenas facilisis, enim quis fermentum rhoncus, nunc magna pretium risus, nec lacinia lectus metus sed tortor. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos himenaeos. Curabitur hendrerit quam sed dolor vehicula pharetra. Morbi pretium ligula vitae nisi ultrices, in mattis nisi mattis. Fusce facilisis ex nec purus malesuada gravida. Quisque facilisis tincidunt sapien, ac dignissim libero porta sit amet. Curabitur in sem et libero imperdiet sollicitudin eget ut turpis. Morbi vulputate risus a augue lacinia, vehicula vestibulum odio aliquet. Donec placerat diam et lorem viverra aliquam. Praesent nibh ipsum, hendrerit eget ultricies mollis, convallis id est. Nullam imperdiet, arcu non faucibus malesuada, metus velit tempor diam, at semper erat urna sed orci.

Acknowledgements

Here goes the acknowledgements.

- Ana - Parity Technologies - Substrate Project

Contents

List of Figures

List of Tables

GLOSSARY

1

Introduction

> *"If Bitcoin was the calculator, Ethereum was the ENIAC*[1]*. It is expensive, slow and hard to work with. The challenge of today is to build the* ***commodity, accessible and performant*** *computer."*
>
> – Unknown.

Blockchains are indeed an interesting topic in 2020. Many believe that they are a revolutionary technology that will shape our future societies, much like the internet and how it has impacted many aspects of how we live in the last few decades (1). Moreover, they are highly sophisticated and inter-disciplinary software artifacts, achieving high levels of decentralization and security, which was deemed impossible so far. To the contrary, some people skeptically see them as controversial, or merely a "hyped hoax", and doubt that they will ever deliver much real value to the world. Nonetheless, through the rest of this chapter and this work overall, we provide ample reasoning to justify why we think otherwise.

In very broad terms, a blockchain is a tamper-proof, append-only ledger that is being maintained in a decentralized fashion, and can only be updated once everyone agrees upon that change as a bundle of transactions. This bundle of transactions is called a **block**. Once this block is agreed upon, it is appended (aka. *chained*) to the ledger, hence the term *block-chain*. Moreover, the ledger itself is public and openly accessible to anyone. This means that everyone can verify and check the final state of the ledger, and all the transactions and blocks in its past that lead to this particular ledger state, to verify everything. At the same time, asymmetric cryptography is the core identity indicator that one needs to interact with the chain, meaning that one's personal identity can stay private in principle, if that is desired based on the circumstances. For example, one's public key

[1] the first generation computer developed in 1944. It fills a 20-foot by 40-foot room and has 18,000 vacuum tubes.

is not revealing their personal identity, whilst using names and email addresses does, like in many traditional systems.

In some sense, blockchains are revolutionary because they remove the need for *trust*, and release it from the control of one entity (i.e. a single bank or institute), by encoding it as a self-sovereign decentralized software. Our institutions are built upon the idea that they manage people's assets, matters and belongings, and they ensure veracity, because we trust in them. In short, they have **authority**. Of course, this model could work well in principle, but it suffers from the same problem as some software do: it is a **single point of failure**. A corrupt authority is just as destructive as a flawed single point of failure in a software is. Blockchain attempts to resolve this by deploying software (i.e. itself) in a transparent and decentralized manner, in which everyone's privacy is respected, whilst at the same time everyone can still check the veracity of the ledger, and the software itself. In other words, no single entity should be able to have any control over the system.

Now, all of these great properties do not come cheap. Blockchains are extremely complicated pieces of software, and they require a great deal of expertise to be written c. Moreover, many of the machinery used to create this *decentralized* and *public* append-only ledger requires synchronization, serialization, or generally other procedures that are likely to decrease the throughput at which the chain can process transactions. This, to some extent, contributes to the skepticism about blockchains' feasibility. For example, Bitcoin, one of the famous deployed blockchains to date, consumes a lot of resources to operate, and cannot execute more than around half a dozen transactions per second (2).

Therefore, it is a useful goal to investigate the possibilities through which a blockchain system can be enhanced to operate *faster*, effectively delivering a higher throughput of transactions per some unit of time.

1.1 Research Questions

We have seen that blockchains are promising in their technology, and unique traits that they can deliver. Yet, they are notoriously slow. Therefore, we pursue the (initial) goal of improving the *throughput* of a blockchain system. By throughput we mean the number of successful transactions that can be processed per some unit of time[1] There are numerous ways to achieve this goal, ranging from redesigning internal protocols within the blockchain to applying concurrency. In this thesis, we precisely focus on the latter, enabling concurrency within transactions that are processed and then appended to the ledger. Moreover,

[1]We will elaborate on the details of *success* and *"unit of time"* in later chapters.

we do so by leveraging, and mixing the best attributes of two different realms of concurrency, namely static analysis and runtime conflict detection [1]. This approach is better compared with other alternatives in section 3.1. We also mention (in the same section) why each of these approaches could have their own merit and value, and how they differ with one another.

Based on this, we formulate the following as our research questions:

RQ1 What approaches exist to achieve concurrent execution of transactions within a blockchain system?

RQ2 How could both static analysis and runtime approaches be combined together to achieve a new approach with minimum overhead and measurable benefits?

RQ3 How would such an approach be evaluated against and compared to others?

1.2 Thesis outline

The rest of this thesis is organized as follows:

write once the outline is done. This is not an important paragraph anyhow.

[1] By static we mean generally anything which is known at the *compile* phase, and by runtime the the *execution* phase

1. INTRODUCTION

2

Background

> *"The use of credit cards today is an act of faith on the part of all concerned. Each party is vulnerable to fraud by the others, and the cardholder in particular has no protection against surveillance."*
>
> – David Chum et. al. - 1990

In this chapter, we dive into the background knowledge needed for the rest of this work. Two primary pillars of knowledge need to be covered: blockchains and distributed systems in section 2.1 and concurrency, upon which our solution will be articulated, in section 2.2.

2.1 Blockchains And Decentralized Applications

In this section, we provide an overview about the basics of distributed system, blockchains, and their underlying technologies. By the end of this chapter, it is expected that an average reader will know enough about blockchain systems to be able to follow the rest of our work and design in chapter 3 and onwards.

2.1.1 Centralized, Decentralized and Distributed Systems

Any introduction to blockchain is always entangled with *distributed* and *decentralized* systems.

A distributed system is a system in which a group of nodes (each having their won processor and memory) cooperate and coordinate for a common outcome. From the perspective of an outside user, most often this is transparent and all the nodes can be seen and interacted with, as if they were *one cohesive system* (3).

Indeed some detail differ between the two, yet the underlying concepts resonate in many ways (4) and blockchains can be seen as another form of distributed systems. Like a distributed system, a blockchain is also consisted of many nodes, operated either by organizations, or by normal people with their commodity computers. Similarly, this *distribution* trait is transparent to the end user, when they want ot interact with the blockchain and they indeed see the system as one cohesive unit.

Blockchains are also **decentralized**. This term was first introduced in a revolutionary paper in 1964 as a middle ground between purely centralized system that have a single point of failure, and a 100% distributed system which is like a mesh (all nodes having links to many other nodes (5) [1]). A decentralized system falls somewhere in between, where no single node's failure can have a unrecoverable damage to the system, and communication is somewhat distributed, where some nodes might act as hops between different sub-networks.

Blockchains, depending on the implementation, can resonate more with either of the above terms. Most often, from a networking perspective, they are much closer to the ideals of a distributed system. From an operational and economical perspective, they can be seen more as decentralized, where the operational power (i.e. the *authority*) falls into the hands of no single entity, yet a large enough group of authorities.

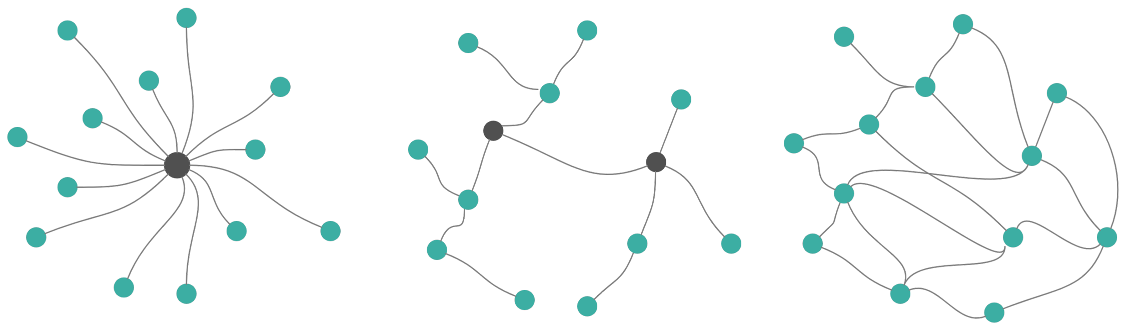

Figure 2.1: Types of network - From left, to right: Centralized, Decentralized, and Distributed.

2.1.2 From Ideas to Bitcoin: History of Blockchain

While most people associate the rise of blockchains with Bitcoin, it is indeed incorrect and the basic ideas of blockchains was mentioned decades earlier. The first relevant research paper was already mentioned in the previous section. Namely, in (5), besides the definition of decentralized system, the paper also describes many other metrics regarding how secure a network is, under certain attacks.

Next, (7) famously introduced what is know as Diffie-Hellman Key Exchange, which is the backbone of public key encryption. Moreover, this key exchange is heavily inspired

[1] The design of Paul Baran, author of (5), was first proposed, like many other internet-related technologies, in a military context. His paper was a solution to the USA's concern about communication links in the after-math of a nuclear attack in the midst of the cold war (6).

by (8), which depicts more ways in which cryptography can be used to secure online communication. Both of these works together form the *digital signature scheme*, which is heavily used in all blockchain systems [1].

Moreover, even the idea of blockchain itself predates Bitcoin. The idea of chaining data together, whilst placing some digest of the previous piece (i.e. a *hash* thereof) in the header of the next one was first introduced in (9). This, in fact, is exactly the underlying reason that a blockchain, as a data structure, can be seen as a append-only, tamper proof ledger. Any change to previous blocks will break the hash chain and cause the hash of the latest block to become different, making any changes to the history of the data structure identifiable, hence *tamper-proof*.

Finally, (10) introduced the idea of using the digital computers as a means of currency in 1990, as an alternative to the rise of credit carts at the time. There were a number of problems with this approach, including the famous double spend problem, in which an entity can spend one unit of currency numerous times. Finally, in 2008 an unknown scientist who used the name Satoshi Nakatomo released the first draft of the Bitcoin whitepaper. In his work, he proposed Proof of Work as a means of solving the double spend problem, among other details and improvements (11). Note that the idea of Proof of Work itself goes back, yet again, to 1993. This concept was first introduced in (12) as means of spam protection in early email services.

2.1.3 Preliminary Concepts

Having known where the blockchain's idea originates from, and which fields of previous knowledge in the last half a decade it aggregates, we can now have a closer look at these technologies and eventually build up a clear and concrete understanding of what a blockchain is and how it works.

2.1.3.1 Elliptic Curve Cryptography

We mentioned the Diffie-Hellman key exchange scheme in section 2.1.2. A key exchange is basically a mechanism to establish a *symmetric* key, using only *asymmetric* data. In other words, two participants can come up with a common shared symmetric key (used for encrypting data) without ever sharing it over the network[2]. Indeed, while the underlying principles are the same, for better performance, most modern distributed systems work with

[1] Many of these works were deemed military applications at the time, hence the release dates are what is referred to as the "public dates", not the original, potentially concealed dates of their discovery.

[2] Readers may refer to (7) for more information about the details of how this novel mechanism works.

another mechanism that is the more novel variant of Diffie-Hellman, namely Elliptic Curve Cryptography, ECC for short. Elliptic Curves offer the same properties as Diffie-Hellman, with similar security measures, whilst being faster to compute and needing smaller key sizes. A key exchange, in short, allows for **asymmetric cryptography**, the variant of cryptography that need not any secrete medium to exchange initial keys, hence it is truly applicable to distributed systems. In asymmetric cryptography, a key *pair* is generated at each entity. A **public** key, which can be, as the name suggests, publicly shared with anyone, and a **private** key that must be kept secret. Any data signed with the private key can be verified using the public key. This allows for integrity checks, and allows anyone to verify the origin of a message. Hence, the private key is also referred to as the **signature**. Moreover, any data encrypted with the public key can only be decrypted with the private key. This allows confidentiality.

Many useful properties can be achieved using asymmetric cryptography, and many massively useful applications adopt it [1]. For blockchains, we are particularly interested in **signatures**. Signatures allow entities to verify the integrity and the origin of any message. Moreover, the public portion of a key, i.e. the public key, can be used as an identifier for each entity.

For example, in the context of banking, a public key can be seen as the account number. It is public and known to everyone, and knowing it does not grant anyone the authority to withdraw money from an account. The private key is the piece that gives one entity *authority* over an account, much like your physical presence at the bank and signing a paper, in a traditional banking system. This is a very common patter in almost all blockchain and distributed systems: Using private keys to sign messages and using public keys as identities.

RSA and DSA are both non-elliptic signature schemes that are commonly known to date. ECDSA, short for **E**lliptic **C**ureve DSA, is the Elliptic Curve variant of the latter. Albeit, ECDSA is a subject of debate, due to its proven insecurities (**?**), and its performance. Hence, more recent, non-patented and open standard [2] curves such as EdDSA are the most commonly used. EdDSA, short for Edwards-curve Digital Signature Algorithm is based on the open standard Edward-curve and its reference, parameters and implementation are all public domain.

[1] The device that you are using to read this line of text has probably already done at least one operation related to asymmetric cryptography since you started reading this footnote. This is how relevant they *really* are.

[2] Unlike ECDSA which is developed and patented by NIST, which in fact is the reason why many people doubt its security.

All in all cryptography and specifically digital signatures play an integral role in the blockchain technology and allow it to operate in the desired way.

2.1.3.2 Hash Functions

Hash functions, similar to elliptic curve cryptography, are among the mathematical backbones of blockchains. A hash function is basically a function that takes some bits of data as input and spits out some bits of output in return. All hash functions have an important property: They will produce a **fixed sized output**, regardless of the input size. Also, a hash function ensures that changing anything in the input, as small as one bit, must result in an entirely different output.

Given this, you can assume that the hash of some piece of data can be seen as its **digest**; If the hash of two arbitrarily large pieces of data is the same, you can assume that their underlying data are indeed the same. This is quite helpful to ensure that some cloned data is not tampered. If we only distribute the hash of the original copy in a secure way, everyone can verify that they have a correct clone, without the need to check anything else.

Albeit, a secure hash function needs to provide more properties. First, the hash function need to ensure that no two different inputs can lead to the same hash. This is called a *collision* and the probability of collision in a hash function should be sufficiently low for it to be of value. Moreover, a hash function must be a *one way* function, meaning that it cannot be reversed in a feasible. By feasibly we always mean: **timely**. If reversing a function takes a few million years, it is *possible*, but not *feasible*. The entire security of hash functions and digital signatures is based on the fact that breaking them is not feasible way [1]. So, Given some hash output, you cannot know the input that lead to that hash. Hash functions that have this property are typically called *cryptographic* hash functions. Cryptographic hash functions are commonly used next to asymmetric cryptography, for authentication and integrity checks, where the sender can sign only a hash of a message and send it over the network, such as the common **M**essage **A**uthentication **C**ode, pattern (13), MAC for short.

2.1.3.3 Peer to Peer Network

From a networking perspective, a blockchain is a purely peer to peer distributed network. A peer to peer network is one in which many nodes form a mesh of connections between them, and they are more or less of the same role and privilege.

[1] And, yes, we are aware of quantum computing, but that is a story for another day.

Remark. A peer to peer network is the architectural equivalent of what was explained as a **distributed** network earlier in this chapter. Similarly, the client-server network model is the equivalent of a **centralized** system.

Unlike a client-server model, a peer to peer network does not have a single point of failure. There is no notion of client and server and all of the entities have the same role, and are simply called a *node*. Having no servers to serve some data, it is predictable to say that peer to peer networks are *collaborative*. A node can consume some resources from another node by requesting some data from it, whilst being the producer for another node by serving some data to it. This is radically different from the client-server model in which the server is always the producer and clients are only consumers, and effectively have no control over the data that they are being served.

Each node in a peer to peer network is constantly talking to other neighboring nodes. For this, they establish communication links between one another. Regardless of the transport protocol (TCP, QUIC (14), etc.), these connections must be secure and encrypted for the network to be resilient. Both elliptic curve cryptography and hashing functions explained in the previous sections, provide the technology needed to achieve this.

In the rest of this work, we are particularly interested in the fact that in a blockchain system, the networking layer provides *gossip* capability. The gossip protocol is an epidemic procedure to disseminate data to all neighboring nodes, to eventually reach the entire network. In a nutshell, it is an *eventually consistent* protocol to ensure that some messages are being constantly gossiped around, until eventually everyone sees them. Blockchains use the gossip protocol to propagate the transactions that they receive from the end user (among many other messages). As mentioned, a distributed system must be seen as a cohesive system from outside, hence, a transaction that a user submits to one node of the network should have the same chance of being appended to the ledger by any of the nodes in the future. Hence, the first requirement is that it must be gossiped around. This becomes more clear when discuss block authoring in section 2.1.3.7.

2.1.3.4 Key-Value Database

Shifting perspective, a blockchain is akin to a database. One might argue that this is too simplistic, but even the brief description that we have already provided commensurate with this. Transactions can be submitted to a blockchain. These transactions are then added to a bundle, called a block, and it is chained with all the previous blocks, forming a chain of blocks. All nodes maintain their view of this chain of blocks, and basically that is what the blockchain is: A database for storing some chain of blocks.

Next to the block database, most blockchains store other data as well to facilitate more complex logic. For example, in Bitcoin, that logic needs to maintain a list of accounts and balances, and perform basic math on top of them[1]. To know an account's balance, it is infeasible to re-calculate it every time from the known history of previous transactions[2]. Hence, we need some sort of persistent database as well to store the auxillary data that the blockchain logic needs, the list of accounts and balances for example. This is called the **state**, and is usually implemented in the form of a key-value database.

A key value database is a database that can be queried similar to *map*. Any value inserted need to be linked with a *key*. This value is then placed in conjunction to a key. The same key can be used to retrieve, update or delete the value. For example, in a Bitcoin-like system, the keys are account identifiers (which we already mentioned are most often just public cryptographic keys), and the values are simply the account balances, some unsigned number.

Indeed, a more complicated blockchain that does more than simple accounting will have a more complicated state layout. Even more, chains that support the execution of arbitrary code (e.g. Smart Contracts (15)), like Ethereum, allow any key-value data pair to be inserted into the state.

One challenge for a nodes in a blockchain network is to keep a **persistent** view of the state. For example, Alice's view of how much money Bob owns need to be the same as everyone else's view. But, before we dive into this aspect, let us first formalize the means of *updating the state*, **transactions**.

2.1.3.5 Transactions and Signatures

So far, we mentioned only transactions to be some sort of information submitted to the system, that are eventually appended to the blockchain in the form of a new block. And, as mentioned, everyone keeps the history of all blocks, essentially having the ability to replay the history and make sur that an account claiming to have certain number of tokens[3] does indeed own it.

But, in the previous section we introduced the concept of *state*, and this is the main reason why transactions exists. Transactions most often cause some sort of update to happen in the state. Moreover, transactions are accountable, meaning that they most often

[1] In reality, Bitcoin does something slightly different, which is known as the UTXO model, which omit to explain here for simplicity.

[2] Imagine an ATM re-executing all your previous transactions to know your current balance every time your query it.

[3] equivalent of a monetary unit of currency, like a coin in the jargon of digital money.

contain a signature of their entire payload, to ensure both integrity and accountability. For example, if Alice wants to issue a `transfer` transaction to send some tokens to Bob, the chain will only accept this transaction if is signed with by alice's private key. Consequently, if there is a fee associated with this transfer, it is deducted from Alice's account. This is where the link between identifiers and public keys also becomes more important. Each transaction has an *origin*, which is basically the identifier of the entity which sent that transaction. Each transaction also has a signature, which is basically the entire (or only sensitive parts of the) payload of the transaction, signed with the private key associated with the aforementioned origin. Indeed, a transaction is valid only if the signature and the public key (i.e. the *origin*) match.

This is a radically new usage of public key cryptography in blockchains, where you can generate a private key using the computational power of your machine and store some tokens linked to it on a network operated by many decentralized nodes, and that private key is the one and only key that can unlock those tokens and spend them. Albeit, while in this chapter we mostly use examples of a cryptocurrency (e.g. Bitcoin), we should note that this is among the simplest forms of transactions. Depending on the functionality of a particular blockchain, its transactions can have specific logic and complexities. Nonetheless, containing a *signature* and some notion of *origin* is very common for most use cases.

Let us recap some facts from the previous sections:

- A blockchain is peer to peer network in which a transaction received by one node will eventually reach other nodes.
- Nodes apply transaction to update some *state*.
- Nodes need to keep a persistent view of the state.

This can easily lead to a race conditions. One ramification of this is the double spend problem: Imagine Eve owns 50 token. she sends one transaction to Alice, spending 40 tokens. Alice checks that Eve has enough tokens to make this spend, update Eve's account balance to 10, basically updating its own view of the state. Now, if Eve sends the same transaction at the same time to Bob, it will also succeed, if Alice and Bob have not yet had time to gossip their local transactions to one another.

To solve this, blockchains agree on a contract: The state can **only** be updated via appending a new block to the known chain of blocks, not one single transaction at a time. This allows to compensate for some potential gossip delays to some extend, and is explained in more detail in the next section.

2.1.3.6 Blocks

Blocks are nothing but a bundle of transaction, and they allow some sort of synchronization, which somewhat relaxes the problem explained in the previous section. To do so, blocks allow nodes to agree on some particular order to apply transaction. For example, in a node, instead of trying to apply transactions that exist on the gossip layer in *some random order*, it will wait to receive a block from other nodes, and then apply them in the same order as stated in the block. transactions inside a block are ordered and applying them sequentially is fully deterministic and will always lead to the same same result. Moreover, in the example of the previous section, it is no longer possible for Eve to spend some tokens twice, because a block will eventually force some serialization of her transaction, meaning that whichever appears second will indeed fail, because the effects of the first one is already apparent and persistent in the state of any node that is executing the block.

A block also contains a small, yet very important piece of data called *parent hash*. This is basically a hash of the entire content of the last know block of the chain. There's exactly one block in each chain that has no parent, the first block. This is a special case and is called the *genesis block*. This, combined with the properties of the hash function explained in 2.1.3.2, bring about the tamper-proof-ness of all the blocks. In other words, the history of operations cannot be mutated. For example, if everyone in the network already knows that the parent hash of the last known block is H_1, it is impossible for Eve to inject, remove, or change any of the previous transaction, because this will inevitably cause the final hash to be some other value, H_2, which in principle should be very different than H_1[1].

All in all, blocks make the blockchain more tamper proof (at least the history of transactions), and bring some synchrony regarding the order in which transactions need to be applied. Nonetheless, with a bit of contemplation, one soon realizes that this is not really solving the race condition, but rather just changing its *granularity*. Instead of the question of which transaction to apply next, we now have the problem of which block to append next. This is because, intentionally, we haven't yet mentioned *who* can propose new blocks to be appended, and *when*. We have only assumed that we *somehow* receive blocks over the network. This brings us to consensus and authorship of blocks, explained in the next section.

[1] In principle, the probability of collision (the hash of some **tampered** chain of blocks being the same as the valid one) is not absolute zero, but it is so small that it is commonly referred to *astronomically small*, meaning that it will probably take millions of years for a collision to happen. As a malicious user, you most often don't want to wait that long.

2.1.3.7 Consensus and Block Authoring

The consensus protocol in a blockchain is constituted of a set of algorithms that ensure all nodes in the network maintain an eventually consistent view of the blockchain (both the chain itself and the state). The protocols need to address problems such as network partition, software failures, and the Byzantine General Problem (16), the sate in which a portion of the nodes in the network *intentionally* misbehave. For brevity, we only focus on one aspect of the consensus which is more relevant to our work, namely, as mentioned at the end of the previous section, the decision of *block authoring*: deciding who can author blocks, and when.

In a distributed system, each node could have a different view of the blockchain, and each node might also have a different set of transactions to build a new block out of (due to the fact that the underlying gossip protocol might have delivered different transactions to different nodes). In principle, any of these nodes can bundle some transactions in a block and propagate it over the network, *claiming* that it should be appended to the blockchain. This will indeed lead to chaos. To solve this, the block authoring is a mechanism to dictate who can author the next block[1]. This must be solved in a decentralized and provable manner. For example, in Proof of Work, each block must be hashed together with a variable in a way that the final hash has a certain number of leading zeros. This is hard to compute, hence the system is resilient against spam. Moreover, this is provable. Any node that receives a candidate block can hash it again and ensure that the block is valid with respect to Proof of Work. In this thesis, the terms "block author", and "validator" are use to refer to the entity that proposes the candidate block, and all the other nodes that validate it, respectively.

The footnote about dictator can be a chapter quote.

Definition 2.1.1. Block **Author**: the network entity that proposes a new candidate block. All other nodes who receive this block and ensure its veracity are referred to as **validators**.

In a Proof of Work scheme, the next author is basically whoever manages to solve the Proof of Work puzzle faster.

Definition 2.1.2. Given the adjustable parameter d, a candidate block data b solving the Proof of Work puzzle is the process of finding a number n such that:

$$Hash(b||n) <= d \tag{2.1}$$

Where d is usually some power of 2 which is equal to a certain number of leading zeros in the output.

[1] In some sense, if blockchains are a democratic system, block authoring is a protocol to chose a *temporary* dictator.

Indeed, this is very slow and inefficient, to the point that many have raised concerns even about the climate impact of the Bitcoin network[1]. There are other consensus schemes such as Proof of Stake, combined with verifiable random functions that solve the same problem, without wasting a lot of electricity.

Nonetheless, we can see how this solves the synchronization issue in blockchains. A block serializes a bundle of transactions. The consensus protocol, namely its block authoring protocol, regulates the block production, so that not everyone can propose candidate blocks at the same time.

2.1.3.8 Interlude: The types of blockchains

So far, we have only talked about *permissionless* blockchains in this work, and we will do so for the rest of the work as well. Nonetheless, now is a good time to mention that a permissionless blockchain is only one variant. Usually blockchains are categorized into 3 types:

- **Permissionless** blockchains: A type of blockchain in which no single entity has any power over the network. Such network are called permissionless, because you need not the permission of an authority to perform an action. For example, as long as you pay the corresponding fee, you can always submit a transaction to a permissionless network, i.e. you cannot be banned by some authority. Or, you can always decide to be a candidate for block authoring, if you wish to do so. Your hardware might not be enough for that to be worthwhile, but you have the freedom to do all of these actions. Such blockchains truly adhere to the decentralized goals of the blockchain ecosystem.

- **Consortium** blockchains: In this type of blockchains, users can still interact with the chain freely, but most *consensus critical* actions are not permissionless. For example, a chain might decide to delegate the task of block authoring to a fixed number of trusted nodes. In such a scenario, none of the mentioned Proof of Work schemes are needed and it can be simplified to a round-robin block authoring. Albeit, such chains are questionable because they don't really solve the main problem of making systems trustless. Such chains are called Proof of Authority, meaning that a node can author a block by by the virtue of being a member of a fixed set of authorities. And from

[1]Some estimates show the annual carbon emission of the Bitcoin network is more than that of Switzerland(17).

Table 2.1: Types of blockchain based on consensus.

	Blockchain Type		
	Public	Consortium	Private
Permissionless?	Yes	No	No
Read?	Anyone	Depends	Invite Only
Write?	Anyone	Trusted Authorities	Invite Only
Owner	Nobody	Multiple Entities	Single Entity
Transaction Speed	Slow	Fast	Fast

the perspective of the end-user, one must still *trust* in the honesty and good will of these authorities.

- **Private** blockchains: these blockchains use the same technology to establish trust between organizations, and are not open to public. A common example would be a chain that maintains government records between different ministries.

It is important to note that many aspects of the consensus protocol, and its complexity will change based on the above taxonomy. The permissionless chains will typically have the most difficult type of consensus, because ensuring veracity is quite hard in a decentralized environment where anyone might misbehave. Albeit, the rationale of the decentralization advocates is that by making the system transparent and open to public, we actually gain more security comparing to hiding it behind servers and firewalls[1], because we can also attract more honest participants that can check the system and make sure it behaves correctly.

Due to all of this complexity, consensus is a very cutting-edge field of research in the blockchain ecosystem. Moreover, in table 2.1 we can already have a glance at how the consensus is also a major factor in the throughput of the blockchain, which is our focus in this work. This correlation is later explained in 3.1.

2.1.3.9 Forks: A Glitch in The Consensus Protocol

Coming back to the permissionless block authoring schemes mentioned in 2.1.3.7, it turns out that a perfect consensus cannot exists in a permissionless network (18). Aside from problems such as a node being malicious and network partitions, there could be other non-malicious scenarios in which everything in the network is seemingly fine, yet nodes end up with different blockchain views. A simple scenario that can lead to this is if, by chance,

[1]One reasonably might see this concept resonating with the Open Source Software movement where an open source software is claimed to be more secure than a closed source one.

two nodes manages to solve the Proof of Work puzzle almost at the same time. They both create a *completely valid* block candidate and propagate it to the network. Some nodes might see one of the candidates, while the others might see another one first. Such scenarios are called a **Fork**: A state in which nodes have been partitioned into smaller groups, each having their own blockchain views. Most consensus protocols solve this by adopting a *longest chain* rule. Eventually, once all block candidates have been propagated, each node choses the longest chain that they can build, and that is be the accepted one. This chain is called the *canonical chain*, and the last block in it is called the *best-block*or the *head* of the blockchain. Based on the canon chain, the state can also be re-created and stored.

Aside from malicious forks (that we do not cover here), and forks due to decentralization such as the example above, there could be *federated* forks as well. For example, if a group of nodes in a blockchain network decide to make a particular change in the history, and they all agree on it, they can simply fork from the original chain and make their new chain. This new chain has some common prefix with the original one, but wi diverges at some point. A very famous example of this is the Ethereum Classic fork from Ethereum network (19). After a hack due to a software bug, a lot of funds got frozen in the main Ethereum network. A group of network participants decided to revert the hack. This was not widely accepted in the Ethereum ecosystem[1] and thus, a fork happened, giving birth to the *Ethereum Classic* network.

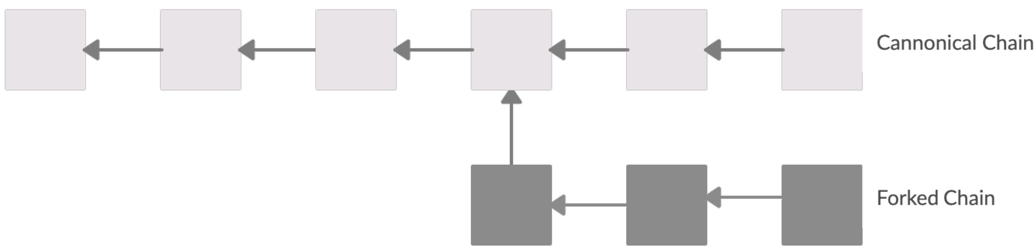

Figure 2.2: Forks - The cannon chain and the forked chain both have a common prefix, yet have *different* best-blocks.

add genesis to the figure

2.1.3.10 Merkle Tree and Storage Root

We already mentioned in 2.1.3.4 that blockchains store some sort of **state** next to their history of blocks as well. Here, we get into more details about this aspect. To recap, the

[1] After all, it defies all the *immutability* properties of a blockchain.

state is a key value database that represents the state of the world, i.e. all the data that is stored besides the history of blocks. States are mapped with block numbers. With each block, the transactions within it could potentially alter the state. Hence, we can interpret this term: "state at block n". This means the final state, given all the blocks from genesis up to n being executed.

First, it is important to acknowledge that maintaining the state seems optional, and it is indeed the case. In principle, a node can decide not to maintain the state and whenever a state value needs to be looked up at a block n, all the blocks from genesis up to n need to be re-executed. This is indeed inefficient. To the contrary, maintaining a copy of the entire state for all the blocks also soon becomes a disk bottleneck. In practice, many chains adopt a middle-ground in which a normal nodes store only the state associated with the last k blocks.

Without getting into too all the details, we continue with a problem statement: In such a database, it is very expensive for two nodes to compare their state views with one another. In essence, they would have to compare *each and every* key value pair individually. To be abel to use this comparison more efficiently, blockchains use a data structure called a Merkle tree[1] (20). A Merkle tree[2] is a tree in which all leaf nodes contain some data, and all non-leaf nodes contain the hash of their child nodes.

There are numerous ways to abstract a key value database with a Merkle tree. For example, one could hash the keys in the database to get a fixed size, base 16, string. Then, each value will be stored at a radix-16 tree leaf which can be traversed by this base 16 hash string.

In such data structure, we can clearly see that the root of the Merkle tree has a very important property: *it is the fingerprint of the* ***entire*** *data.* This piece of data is very important in blockchains as is usually referred to as **state root**. In essence, if two nodes compute their individual state roots and compare them, this comparison would confidently show if they have the same state or not. This is very similar to how the existence the parent hash in each block ensures veracity that all nodes have the same block prefix: changing only a bit in a previous block, or a state value in this case, will cause the hashes to no longer match. Similarly, changing only one value in the entire key-value database will cause the state roots to mismatch.

Recalling the definition of author and validator from 2.1.1, we can now elaborate more on what a validator exactly does. A validator node, upon receiving a block, should check

[1] Sometimes referred to as "Trie" as well.

[2] Named after Ralph Merkle, who also contributed to the foundation of cryptography in (8).

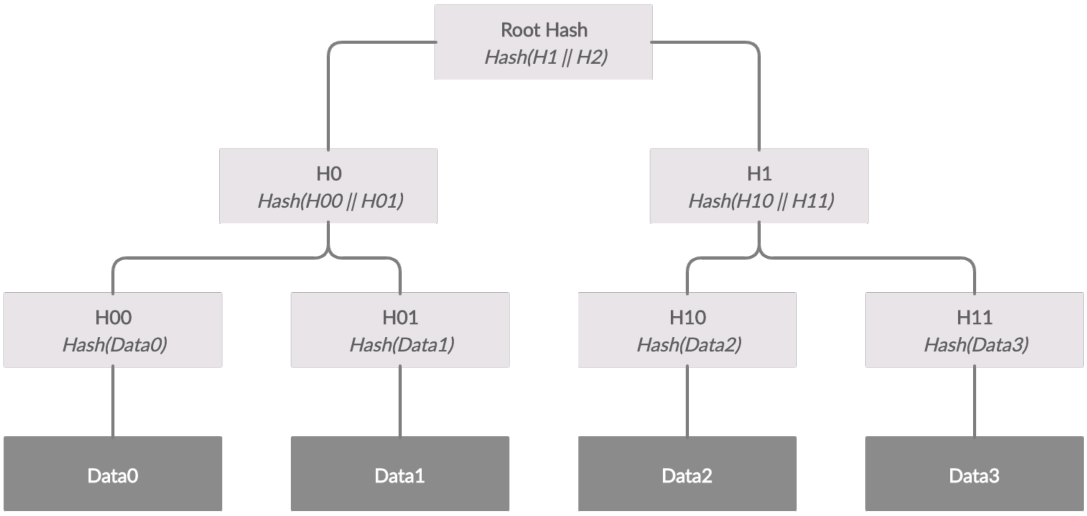

Figure 2.3: Merkle Tree - A binary Merkel Tree. The root hash contains a digest of all the 4 data nodes.

that the block's author is valid (for example check the proof of work puzzle), and then it re-executes all thr transactions in the block, to compute its own state root. Finally, this state root is compared with the state root that the block author proposed in the block, and if they match, the block is valid.

We can now summarize all the common data that are usually present in a block's header:

- Parent hash: As mentioned, this is the signature of the blockchain prefix.
- Block number: A numeric representation of the block count, also known as blockchain *height*.
- State root: Finally, it is common for a block to also name the state root that should be computed, if the transaction inside the block body are executed on top of the aforementioned parent hash block's state.

Our definition of the basic concepts of *blockchain protocols* almost ends here. In the next sections, we briefly explain more concepts that are more relevant to an implementation of a blockchain, not the protocol itself.

2.1.3.11 Runtime

With all that being said, we will coin the term *Runtime* as the piece of logic in the blockchain that is responsible for *updating the state*. To be more specific, the runtime of any blockchain can be simplified as a simple function that takes a transaction as input, has access to read the state, and as output generates a set of new key-value pairs that need to be updated in the state (or the runtime itself can update the state directly, depending on the design of the system). This abstraction will be further used in chapter 3.

2.1.3.12 Transaction Pool

By defining the transaction pool, we will distinguish transactions that are *included* in any block, and those that are not. As mentioned, a blockchain node might constantly receive transactions, either directly from end-users, or from other nodes as their role in some sort of gossip protocol. These transactions are all pending, and their existence does *not* imply anything about the state of the blockchain. Only when, by some means of consensus, everyone agrees to append a block to the chain, then the transactions within that block are included in the chain. Hence, transaction can be categorized into *included* and *pending*.

The transaction pool is the place where all the *pending* transactions live in. Its implementation details are outside the scope of this work and depend on the needs of the particular chain. Nonetheless, we highlight the fact that the transaction pool is a component that sits next to the block authoring process. Once a node wants to author a block (or just try to do so, in cases such s Bitcoin where some Proof of Work puzzle need to be solved first), it will use the transactions that it has received and have been stored in the transaction pool as a source of block building.

2.1.3.13 Transaction Validation

Usually, a transaction need to pass some bare minimum checks to even be included in the pool, not to mention being included in the canonical chain. Usually, checks that are mandatory, persistent and rather cheap to compute can happen right when a transaction is being inserted in the pool. For example, the signature of a transaction must always be valid and its validity status persist over time. In other words, if the signature is correct, it will *stay* correct over time. To the contrary, state-dependent checks usually need to be performed when a transaction is being *included*, not when it is being inserted to the pool. The reason for this is subtle, yet very important. If a transaction is asserting to transfer some tokens from Alice to Bob, the state dependent check is to make sure Alice has enough tokens. In principle, it is wrong to check Alice's account balance at the time of inserting the transaction into the pool, since we *do not know* when this transaction is going to be *included*. What matters is that *at the block in which this transaction is being included*, Alice must have enough tokens.

That being said, an implementation could optimise the read from state in some particular way to allow more checks to happen in the transaction pool layer (one if which explained in the next section, 2.1.3.14). Although, it should be noted that transactions in the pool are not yet *accountable*, since they are not executed. In other words, a user does not pay any fees to have their transaction live in the pool. But, they do pay to have their transaction included in the chain. Therefor, if the pool spends too much time on validation, this can easily turn into a **Denial of Service** attack, DoS for short.

2.1.3.14 Account Nonce

We mentioned that signatures allow transactions to be only signed only by an entity that owns a private key associated with the account. This allows anyone to verify a transaction that claims to spend some funds from an account. Nonetheless, given that the block history is public, this pattern is vulnerable to *replay attacks*. A replay attack is attack in which a malicious user will submit some (potentially signed) data twice. In the case of a blockchain, Eve can simply lookup a transaction that transfers some money out of Alice's account, and re-submit it back to the chain numerous times. This is an entirely valid operation by itself, since the transaction that Eve is submitting again indeed does contain a valid signature from Alice's private key.

To solve this, blockchains that rely on state usually introduce the concept of nonce: a counter that is associated with each account in state, initially set to zero for every potential account. A transaction is only valid if in its signed payload, it provides the nonce value associated with the origin, incremented by one. Once the transaction is included, the nonce of the account is incremented by one. This effectively alleviates the vulnerability of reply attacks. Any transaction that Alice signs, once submitted to the chain and upon being *included*, is no longer valid for re-submission.

This needs to be mentioned in the later chapter that we don't specifically deal with it.

2.1.3.15 Putting it All Together: Decentralized State Transition Logic

We close our introduction to blockchains with providing a final perspective on their nature. First, we enumerate some of the lenses through which we have seen blockchains:

- A distributed peer to peer network of nodes.
- A distributed database of transactions.
- A decentralized trustless transaction processing unit.

We can put all of this together into one frame by representing blockchains as **state machines**. This concept resonates well with our notion of state database as well. A blockchain is a *decentralized* state machine. It is a state machine because the state-root hash at the end of each block is one potential state, and blocks allow for transition between states. Due to forks, one might have to revert to a previous state. It is decentralized because there is no single entity that can enforce transition from one state to another one. In fact, any participant can propose transitions by authoring a block candidate, but it will only ever be considered canon if it is agreed upon by everyone, through the consensus mechanism. Moreover, each participant stores a copy of the state. If a single node crashes, goes offline, or decides to misbehave, the integrity of the system is maintained, as long as there are enough honest participants.

2.1.4 Disclaimer: A Note About The Context of Technology

Before continuing with the chapter, we briefly address the issue *Technology Context*. So far in this chapter, we have used simple examples from the banking world, since it is similar to Bitcoin which is a very known system and it is easy to explain. Nonetheless, a reader who may have previously had some background knowledge with some other blockchain project X might soon find some details that we named here and they might not be 100% compatible with project X. Moreover, we have even admitted throughout the text that some of our examples are not even exactly similar to Bitcoin (such as the state model as opposed to the UTXO model).

This is entirely predictable to happen, as blockchain systems are a rapidly evolving field of science and technology at the moment. Different project diverge from one another, even in radical concepts, and experiments with new patterns. Nonetheless, we make the following assertions about our assumptions in this work:

- Whenever we build up a simple example (mostly with Alice and Bob) in this work, we do not tie it to any particular blockchain project. Instead, these examples are to be interpreted completely independent and solely based on the relevant concepts explained.
- As for the rest of the text, including the earlier sections of this chapter, we will attempt to see blockchains in the most *generic* form that they can be seen. That is to say, we interpret blockchains exactly as we defined in 2.1.3.15: A decentralized state machine that can be transitioned through means of any form of opaque transaction.

To summarize, in this chapter, we have explained only what we have deemed to be the most fundamental details of blockchains, and we noted whenever a detail could potentially be different based on implementation. This approach will persist throughout the rest of this work as well.

2.2 Concurrency

In this section we introduce some relevant concepts from the field of concurrency. As mentioned, the crux of our idea is to deploy concurrency in a blockchains system to gain throughput.

Concurrency is the ability of a software artifact to execute units of its logic out of order, without causing the outcome be invalid. We will link this directly to our example of interest: A node in a blockchain system has a process which is responsible for executing blocks. By default this process is purely sequential: All of the transactions in the block are executed in sequence. Deploying concurrency *should* allow this process to divide the transactions within the block into an arbitrary number of smaller groups, each of which can be executed asynchronously and and out of order, without causing an invalid outcome.

The outcome of interest, of course, is only the aforementioned **state database**. Nodes in the network receives blocks and apply them to a common state. The only acceptable outcome is for all of them to come up to the same state after applying the block. As mentioned, this comparison is done by the means of state root.

Definition 2.2.1. Valid Block: A block is valid only if its execution is deterministic among all of the nodes in the network, and leads to the same state root $S^{'}$ if applied on top of a fixed state S.
Another

This is in fact the property that ensures that the state is, in principle, optional to keep around, because it is deterministically recreatable from the chain of blocks.

Thus, the **need for determinism is absolute** in a blockchain's runtime environment. Moreover the previous remark sheds some light into why blockchains are designed to work sequentially by default: Because it is easy to ensure determinism when all the transactions are applied sequentially.

With this, we can reduce the problem to a single node's hardware. Assume a node is attempting to execute a block and it has the entire state loaded in memory. If a single thread executes the transactions within the block, it is trivial to show that the outcome is deterministic. On the other hand, one can easily see that if multiple threads try to execute the transactions concurrently and access the state as they move forward, the result is unworthy and moot. The challenge is to allow these threads to cooperate and achieve an effective concurrency, while still maintaining determinism. Therefor, we have translated out blockchain scenario in a typical shared state concurrency problem[1]. In such a setup, multiple threads are competing for access to some shared data (the state) and the runtime environment need to resolve the race conditions between the threads.

In the next sections we will look at practical ways to use *concurrency over a shared state*, while still generating valid results. A mechanism that provides this is called *concurrency control*.

2.2.1 Locking: Pessimistic Concurrency Control

Locks are among the simples and most intuitive ways to handle concurrency. A lock is an abstract notion applied to a shared object (or generally any memory region) which limits the number of threads that can access it at the same time in a multiprocessor system. The idea of using locks in database systems that want to achieve higher throughput by using multiple threads goes back to many decades ago (21, 22).

The simplest form of a lock will not distinguish between reads and writes and the only operation on it are `acquire` and `release`. Once a thread acquires a lock, no other thread can, and they have to wait for it. Once the holding thread is done with the lock (more accurately done with the data protected by the lock), they release the lock. Upon being released, one of the waiting threads acquires the lock and so on. This process can easily ensure that some data is never accessed by multiple threads at the same time, The processor usually ensures that these primitive operations (`acquire` and `release`) are done atomically between threads. Such locks that do not distinguish between read and writes are called a `Mutex`, short for mutual exclusion (23).

A more elaborate variant of Mutex is read-write locks, RW locks for short. Such locks leverage the fact that multiple reads from the same data are most often harmless and should be allowed. Thus, RW locks distinguish in between them. In a RW lock, at any given time there can be only one write locks,but multiple concurrent read locks are allowed.

Remark. We use the Rust programming language for the implementation of this work. Rust provides some of the finest compile-time memory safety guarantees among all languages. To achieve this, Rust references (addresses to memory locations) have the exact same aliasing rule: multiple immutable references to a data can co-exits in a scope, but only one mutable reference is allowed (24).

[1]With the remark that the determinism requirement is somewhat special to our use case and some system might not require it.

In some sense, locks are a pessimistic means of concurrency control. They assume if two threads want to acquire the same lock, their logic *will* cause a conflict to happen. Based on the granularity of the lock and the internal logic of each threads, this might not be the case all the time.

2.2.2 Transactional Memory: Optimistic Concurrency Control

Transactional memory is the opposite of locking when it comes to waiting. In locking, the threads often need to wait for one another. If a thread is writing, then all the readers and writes need to wait. As mentioned, this is based on the assumption that mutual acquirement of locks will always lead to conflicts, so it need to be prevented in any case. Transactional memory takes the opposite approach and assumes that mutual data accesses will not conflict by default. In other words, threads don't need to wait for one another, thus transactional memory is coined "lock free" (25).

In the context of transactional memory, a thread's execution is divided into smaller pieces of logic called transactions. A transaction attempts to apply one or more updates to some data, without waiting and then *commits* the changes. Before a commit, the changes by any transaction are not visible to any other transaction. Once a commit is about to happen, the runtime must check if these changes made by it are applicable, based on the previous commits. If committed successfully, the changes are then visible to all other transactions. Else, none of the changes become visible and the transaction *aborts* and the changes are reverted.

Remark. Software transactional memory, as the name suggests, is usually used in transaction systems where a transaction is the unit of logic that alters the data, much like a blockchain or a database. Therefore, usually the term "transaction" is favoured in the literature. In this section we also use the work "transaction" for consistency, but it can generally be interchanged with "thread".

Transactional memory can exist either via a specialized hardware or, simulated in the software, referred to as *software* transactional memory, or STM (26). As the *"software"* prefix denotes, transactional memory is typically implemented in the software and does incur a runtime overhead. Nonetheless its programming api is much easier and less error prone (27).

Transactional memory is likely yo cause less conflicts, nonetheless it still allows threads to operate over a same data structure. This implies complications about commits, aborts, and coherence[1]. A radical different approach is to try and prevent these complications from the get go, by disallowing the shared data to exist, which is described in the next section.

2.2.2.1 Note About Determinism

It is very important to note that both locking and transactional memory are non-deterministic. This means that executing the same transactions multiple time may or may not lead to the same output. It is easy to demonstrate why: Imagine two threads will soon attempt to compete for a lock, and each will try and write a different value to a protected memory address. Based on the fairness rules of the underlying operating system and sheer randomness of which thread executes the acquire faster, either of them could be the first one. While the output of the program in both cases is *correct*, it is not deterministic.

The same can be said about transactional memory. If two transaction have both altered the same data in a conflicting way, one of them is doomed to fail upon trying to commit, and it just the matter of which will do it first. Again, both outputs are correct, yet the program is not deterministic.

Remark. A read can at this point link this insight into our requirement of a block being validated: From the perspective of a block author, it does not matter what the outcome of a block is (i.e. which transactions are within it, which succeed and which fail). The first and foremost importance is for all validators to execute the same outcome (i.e. come to the same state root), once having received the block later. This is in stark contrast with the non-determinism nature of locking.

2.2.3 Sharing vs. Communicating

There is a great quote from the documentation of the Go programming language that greatly explains the point of this section: "Don't communicate by sharing memory; share memory by communicating. (R. Pike)" (28). This introduces a radical new approach to concurrency, in which threads are either stateless or pure functions (TODO: ref), where their state is private. All synchronization is then achieved by the means of message passing. Instead, threads don't need to share any common state or data. If threads need to all manipulate the same data, they can send references to the data to one another. In many cases, this pattern is advantageous compared to locking both in terms of concurrency degree and the programming ease.

[1] The question of which changes from a local thread's transaction become visible to other threads and when, under which conditions. We have barely touched this issue and in itself deserves a thesis to be fully understood.

Nonetheless, we know that our use case is exactly the need for some executor threads to have a shared state while executing the blockchain transactions. Therefor, we won't directly apply this paradigm, but we inspire from it and take the possibility of message passing into account. We will revisit this possibility in chapter 3.

2.2.4 Static Analysis

As mentioned in 2.2.2, transactional memory attempts to reduce the waiting time by assuming that conflicts are rare. This could bring about two downsides: Reverts, in case a conflict happens, and the general runtime overhead that the system need to tolerate for all the extra machinery needed for transactional memory[1]. An interesting approach to counter these limitation is static analysis. Static is referred to any action that is done at compile time, contrary to runtime. The goal of these static analyses is to somehow improve the concurrency degree. In the case of transactional memory, this could be achieved by using static analysis to aborts(29). Similarly, other studies have tried to use static analysis to improve the usage of locking by automatically inserting the lock commands into the program's source code at compile time (30). This can greatly ease the user experience of programmers using locks and reduce the chance of human errors possibility[2].

[1]Which is even more if it is being emulated in the software.

[2]Needless to say, concurrent programming is dauntingly hard, specially if only primitive tools such as locks can be used. One of the main purposes of STM is the move this burden of the shoulders of programmer.

3

System Design

introductory paragraph explaining the outline of the chapter:
1. We will first talk about what different ways to speed up a blockchain as an prelude. Last item in this list will be concurrency.
2. Then we will talk more about concurrency and different ways that it can be achieved, like a mini *related work* section, but I will not go into too much detail and spare that for later.
3. Based on this related work, I point out some deficiencies in them and come up with design requirements.
4. Then finally I propose my design which addresses everything that I said is required.

At this point I should really define what is my metric. Thrhoughput? efficiency? what?
I am currently way more in favour of only using throughput (tx/sec).

3.1 Prelude: Speeding up a Blockchain - A Brief Overview

In this section, we briefly survey some of the ways through which the throughput of a blockchain can be improved, and delineate which approach we focus on, for the rest of this work.

As mentioned, blockchains can be seen, in a very broad way, and from a transaction processing point of view, as a *decentralized state machine that transitions by means of transactions*. The throughput of a blockchain network, measured in transactions per second, is a function of numerous components and can be analysed from different points of view [1]. While in this work we focus mainly on one aspect, it is helpful to enumerate different viewpoints and see how each of them affect the overall performance.

3.1.1 Consensus and Block Authoring

As mentioned, the consensus protocol provides the means of ensuring that all nodes have a persistent view of the state, and it can heavily contribute to the throughput of the system. Two common consensus protocols are Proof of **Work** and Proof of **Stake**. They use the computation power (*work*) and a number of bonded tokens (*stake*), respectively, as their guarantees that an entity has *authority* to perform some operation, such as authoring a block. It is important to note that each of these consensus protocols has an *inherently*

[1]and this categorization is by no means exhaustive.

different performance (?). Proof of work, as the names suggests, requires the author to prove their legitimacy by providing a proof that they have solved a particular hashing puzzle. This is slow by nature, and wastes a lot of computation power on each node that wants to produce blocks, which in turn has a negative impact on the transaction throughput. Speeding-up this metric requires the network to agree on an easier puzzle, that can in turn make the system less secure (?). More precisely, the difficulty of the puzzle dictates the average time any node needs to spend to be able to produce a block, which dictates the final throughput.

To the contrary, Proof of Stake does not need this this puzzle solving, which is beneficial in terms of computation resources. Moreover, since the chance of any node being the author is determined by their stake. In a such a authoring process, a smaller block-time is not insecure by itself. Even more recent, we are seeing blockchains turning into verifiable random functions (?)[1] for block authoring, and deploy a traditional byzantine fault tolerance voting scheme on top of it to ensure finality. This further decouples block production and finality, allowing production to proceed faster and and with even less drag from the rest of the consensus.

All in all, one general approach towards increasing the throughput of a blockchain is to *re-think the consensus and block authoring mechanisms* that dictate when blocks are added to the chain, and by whom, with what frequency. It is crucially important to note that any approach in this domain falls somewhere in the spectrum of centralized-decentralized, where most often approaches that are more centralized will be more capable of delivering better throughput, yet they may not have some of the security and immutability guarantees of a blockchain. An example of this is provided in table 2.1 and private blockchain.

3.1.1.1 Sharding

An interesting consensus-related optimisation that is gaining a lot of relevance in the recent year is a technique, obtained from the databases field, called *sharding*. Shards are slices of data (in the database jargon) that are maintained on different nodes. In a blockchain system, shards refer to sub-chains are are maintained by sub-networks of the whole system. In essence, instead of keeping all the nodes in the entire system synchronized at all times, *sharded blockchains* consist of multiple smaller networks, each maintaining their own cannon chain. Albeit, most of the time these chains all have the same prefix and only differ in the most recent blocks. At fixed intervals, sub-networks come to agreement

[1] VRF for short.

and synchronize their shards with one another. In some sense, sharding allows smaller sub-networks to progress faster. Much work has been done in this field (**? ? ?**).

3.1.2 Chain Topology

Another approach is changing the nature of the chain itself. A classic blockchain is theoretically limited due to the fact that only one entity can append a block to the chain at each time. This property will bring extra security, and make the chain state easier to reason about (i.e. there is only one cannon chain). A radical approach is to question this property and allow different blocks to be created at the same time. Consequently, this approach turns a blockchain from a literal *chain of blocks* into a *graph*. Most often, such technologies are referred to Directed Acyclic Graphs, **DAG** in short, solutions.

Altering chain topology will bring even more radical changes to the original idea of blockchain. While being very promising for some fields, such as *Internet of Things* and micro payments, we will not consider DAGs in this work: we choose to adhere to the definitions provided in the previous sections as our baseline of *what a blockchain is*.

3.1.3 Deploying Concurrency over Transaction Processing

Finally, we can focus on the transaction processing view of the blockchain, and try and deploy concurrency on top of it, leaving the other aspects such as consensus unchanged and, more importantly, generic. This is very important, as it allows our approach to be deployed on many chains, since it is independent of many consensus and network details. We have already explained how a block author proposes a new block, and how all validators must re-execute the transactions to ensure validity in the previous chapters. Thus, a block's lifecycle can be enumerated into two different phases:

- Block authoring: done by one block author.
- Block validation: done by all the nodes in the network.

By default, these phases must happen in a **sequential** manner, to ensure *consistency*. In other words, to ensure that the validators will come to the same state root as the one proposed by the block author.

Our work specifically focuses on this aspect of of the blockchain systems and proposes a novel approach to achieve concurrency within each block's execution, both in the authoring phase and in the validation phase, thereby increasing the throughput.

3.1.4 Analysis of the approaches and their merits

Indeed, it is questionable how much gain will concurrency bring to the overall throughput. To counter this doubt, we argue that the aim of this work is to foremost improve the *efficiency* of the transaction processing power of the blockchain. It is true that in a specific chain, the network or consensus might be the main bottleneck of the throughput, and concurrency cannot improve that by much. For example, most performance gains within concurrency might be within milliseconds, while the block authoring delay is tens of seconds in some chain. Nonetheless, most modern hardware have built-in features to run code in parallel, which translates to adopting more concurrency on the application layer.

Therefore, we argue that our method is *universal*, and independent of the underlying chain specifications, as long as it adheres to our basic definitions in chapter 2, such as having a state root, and authors and validators per block. And, despite its degree of impact, concurrency can be applied to any chain. For some chains, it might be a dominant factor and drastically improve the *throughput*, whilst in others it might only contribute to the *efficiency* of the system and allow the blockchain to better use the hardware that is provided to it[1].

3.2 Concurrency Within Block Production and Validation

This is where I will talk more about different approaches done by others within th realm of concurrency. I will argue that there are two main appraoches, runtime and static and how each one totally misses the benefits of the other one.

This section will answer research question 1.

We finish this section by setting a few design requirements:

1. The system should still be deterministic. i.e. anything done by authors will have the same impact on validators. This is the single most important requirement.

2. The system should always terminate (no deadlock), and have as little contention as possible.

3. The system should not incur too much overhead on validators (i.e. I don't like dependency graphs.)

[1]It is worth noting that having optimal hardware utilization (to reduce costs) is an important factor in the blockchain industry, as many chains are ran by people who are making profit out of running validators and miners.

3.3 System Design

Basically https://www.notion.so/Architecture-V1-cfe987d4dd4345569b07649dcf00a85c

This section will answer research question 2.

4

Implementation

Just detail of the implementation.

5

Related Work and Discussion

...

In this chapter we will survey the related works in the filed and discuss how our approach differs from them.

5.1 Related Work

5.2 Discussion

6

Conclusion

...

Final words and conclusions.

6.1 Conclusion

6.2 Further Work

Appendix

References

[1] Craig Pirrong. **Will Blockchain Be a Big Deal? Reasons for Caution**. *J. Appl. Corp. Finance*, **31**(4):98–104, 2019. 1

[2] Arthur Gervais, Ghassan O. Karame, Karl Wüst, Vasileios Glykantzis, Hubert Ritzdorf, and Srdjan Capkun. **On the Security and Performance of Proof of Work Blockchains**. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 3–16, Vienna, Austria, October 2016. Association for Computing Machinery. 2

[3] IMRAN BASHIR. *MASTERING BLOCKCHAIN: Distributed Ledger Technology, Decentralization, and Smart Contracts Explained, 2nd Edition;Distributed Ledger.* PACKT Publishing, Place of publication not identified, 2018. 5

[4] Maurice Herlihy. **Blockchains from a Distributed Computing Perspective**. *Commun. ACM*, **62**(2):78–85, January 2019. 5

[5] P. Baran. **On Distributed Communications Networks**. *IEEE Trans. Commun. Syst.*, **12**(1):1–9, March 1964. 6

[6] 1776 Main Street Santa Monica and California 90401-3208. **Paul Baran and the Origins of the Internet**. https://www.rand.org/about/history/baran.html. 6

[7] W. Diffie and M. Hellman. **New Directions in Cryptography**. *IEEE Trans. Inf. Theory*, **22**(6):644–654, November 1976. 6, 7

[8] Ralph C. Merkle. **Secure Communications over Insecure Channels**. *Commun. ACM*, **21**(4):294–299, April 1978. 7, 18

[9] Stuart Haber and W. Scott Stornetta. **How to Time-Stamp a Digital Document**. *J. Cryptology*, **3**(2):99–111, January 1991. 7

[10] David Chaum, Amos Fiat, and Moni Naor. **Untraceable Electronic Cash**. In Shafi Goldwasser, editor, *Advances in Cryptology — CRYPTO' 88*, Lecture Notes in Computer Science, pages 319–327, New York, NY, 1990. Springer. 7

[11] Satoshi Nakamoto. **Bitcoin: A Peer-to-Peer Electronic Cash System**. page 9. 7

[12] Cynthia Dwork and Moni Naor. **Pricing via Processing or Combatting Junk Mail**. In Ernest F. Brickell, editor, *Advances in Cryptology — CRYPTO' 92*, Lecture Notes in Computer Science, pages 139–147, Berlin, Heidelberg, 1993. Springer. 7

[13] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. **Keying Hash Functions for Message Authentication**. In Neal Koblitz, editor, *Advances in Cryptology — CRYPTO '96*, Lecture Notes in Computer Science, pages 1–15, Berlin, Heidelberg, 1996. Springer. 9

[14] Gaetano Carlucci, Luca De Cicco, and Saverio Mascolo. **HTTP over UDP: An Experimental Investigation of QUIC**. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, SAC '15, pages 609–614, Salamanca, Spain, April 2015. Association for Computing Machinery. 10

[15] **Ethereum/Wiki**. https://github.com/ethereum/wiki. 11

[16] Leslie Lamport, Robert Shostak, and Marshall Pease. **The Byzantine Generals Problem**. *ACM Trans. Program. Lang. Syst.*, **4**(3):382–401, July 1982. 14

[17] Christian Stoll, Lena Klaassen, and Ulrich Gallersdörfer. **The Carbon Footprint of Bitcoin**. *Joule*, **3**(7):1647–1661, July 2019. 15

[18] Wenbo Wang, Dinh Thai Hoang, Peizhao Hu, Zehui Xiong, Dusit Niyato, Ping Wang, Yonggang Wen, and Dong In Kim. **A Survey on Consensus Mechanisms and Mining Strategy Management in Blockchain Networks**. *IEEE Access*, **7**:22328–22370, 2019. 16

[19] Paul Vigna. **The Great Digital-Currency Debate: 'New' Ethereum Vs. Ethereum 'Classic'**, August 2016. 17

[20] RALPH C. MERKLE. **A Digital Signature Based on a Conventional Encryption Function**. In CARL POMERANCE, editor, *Advances in Cryptology — CRYPTO '87*, Lecture Notes in Computer Science, pages 369–378, Berlin, Heidelberg, 1988. Springer. 18

[21] ZVI KEDEM AND ABRAHAM SILBERSCHATZ. **Controlling Concurrency Using Locking Protocols**. In *20th Annual Symposium on Foundations of Computer Science (Sfcs 1979)*, pages 274–285, October 1979. 24

[22] R. J. T MORRIS AND W. S WONG. **Performance Analysis of Locking and Optimistic Concurrency Control Algorithms**. *Performance Evaluation*, **5**(2):105–118, May 1985. 24

[23] RACHID GUERRAOUI, HUGO GUIROUX, RENAUD LACHAIZE, VIVIEN QUÉMA, AND VASILEIOS TRIGONAKIS. **Lock–Unlock: Is That All? A Pragmatic Analysis of Locking in Software Systems**. *ACM Trans. Comput. Syst.*, **36**(1):1:1–1:149, March 2019. 24

[24] AARON WEISS, OLEK GIERCZAK, DANIEL PATTERSON, NICHOLAS D. MATSAKIS, AND AMAL AHMED. **Oxide: The Essence of Rust**. *ArXiv190300982 Cs*, August 2020. 24

[25] TOM KNIGHT. **An Architecture for Mostly Functional Languages**. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, LFP '86, pages 105–112, New York, NY, USA, August 1986. Association for Computing Machinery. 25

[26] LANCE HAMMOND, VICKY WONG, MIKE CHEN, BRIAN D. CARLSTROM, JOHN D. DAVIS, BEN HERTZBERG, MANOHAR K. PRABHU, HONGGO WIJAYA, CHRISTOS KOZYRAKIS, AND KUNLE OLUKOTUN. **Transactional Memory Coherence and Consistency**. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, ISCA '04, page 102, USA, March 2004. IEEE Computer Society. 25

[27] MAURICE HERLIHY AND J. ELIOT B. MOSS. **Transactional Memory: Architectural Support for Lock-Free Data Structures**. *SIGARCH Comput. Archit. News*, **21**(2):289–300, May 1993. 25

[28] **Share Memory By Communicating - The Go Blog**. https://blog.golang.org/codelab-share. 26

[29] Ricardo J. Dias, João M. Lourenço, and Nuno M. Preguiça. *Efficient and Correct Transactional Memory Programs Combining Snapshot Isolation and Static Analysis*. 27

[30] Sigmund Cherem, Trishul Chilimbi, and Sumit Gulwani. **Inferring Locks for Atomic Sections**. August 2007. 27