Vrije Universiteit Amsterdam

Universiteit van Amsterdam

Master's Thesis

SonicChain: A Lock-free, Pseudo-Static Approach Toward Concurrency in Blockchains

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

October 18, 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: In Praise of Web3 and the Decentralized Web Movement

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 FIGURES

List of Tables

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 (e.g. a single bank, an institute, or simply Google), 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

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

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, 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

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 Distributed Ledger Technology

In this section, we provide an overview about the basics of distributed systems, 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 understand the approach proposed 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 own 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 details 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 to 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 systems that have a single point of failure, and 100% distributed systems, which are 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 irrecoverably damage 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.

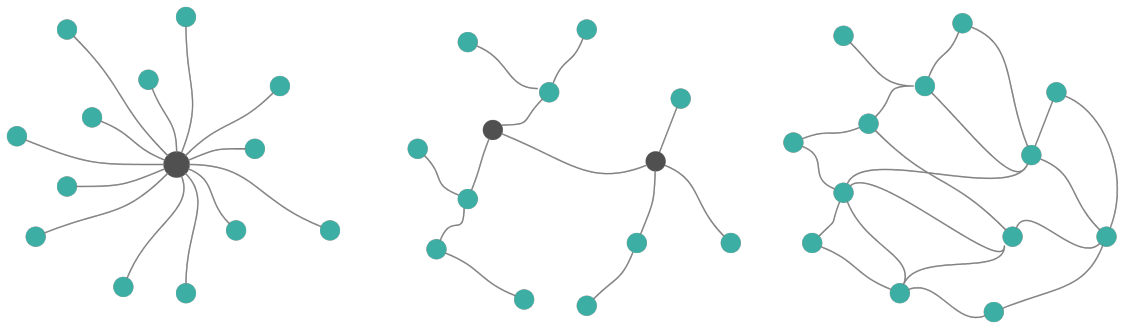

Figure 2.1: Types of networks. - From left to right: Centralized, Decentralized, and Distributed.

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 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, 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 an 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 cards 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

[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.

with another mechanism that is the more novel variant of Diffie-Hellman, namely Elliptic Curve Cryptography (ECC). 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 no secrete medium to exchange initial keys, and therefore 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 pattern 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 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, does result in an entirely different output.

Given these properties, one 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 with. 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 needs to ensure that no two different inputs can lead to the same hash. This - the situation that different inputs generate the same hash - 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 manner. By feasible we effectively 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 [1]. So, given some hash output, one 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).

2.1.3.3 Peer to Peer Networks

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, being simply called *nodes*. 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 data from it, whilst being the producer for another node by serving data to it. This is radically different from the traditional 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. Thus, 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. Consequently, the first requirement is that it must be gossiped around. This becomes more clear when we 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 commensurates with this. Transactions can be submitted to a blockchain. These transactions are then added to a bundle, called a block, which is then 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]. Thus, we need some sort of persistent database as well, to store the auxiliary data that the blockchain logic needs - like, the list of accounts and their balances in our 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 in the database needs 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 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 needs 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*: the **transactions**.

2.1.3.5 Transactions and Signatures

So far, we mentioned transactions to be some sort of pieces 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 sure that an account claiming to have a certain number of tokens[3] does indeed own it.

The concept of *state* is the main reason why transactions exist. Transactions most often cause some sort of update to happen in the state. Moreover, transactions are accountable, meaning that they most often contain a signature of their entire payload, to ensure both

[1] In reality, Bitcoin does something slightly different, which is known as the UTXO model, which we 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] Tokens are the equivalent of a monetary unit of currency, like a coin in the jargon of digital money.

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 the 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 one can generate a private key using the computational power of a personal machine, and store some tokens linked to it on a network operated by many decentralized nodes; that private key is the one and only key that can unlock those tokens and spend them. Although 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 complexity. 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 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, updates 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 potential gossip delays to some extent, and is explained in more detail in the next section.

2.1.3.6 Blocks

Blocks are nothing but bundles 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 transactions. For example: a node, instead of trying to apply transactions that exist on the gossip layer in *some random order*, 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: it 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 transactions, meaning that whichever appears second will indeed fail, because the effects of the first one are 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 is 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. *Authoring and Validating.*
Author: the network entity that proposes a new candidate block. This task is called block *authoring*.
Validator: All other nodes who receive this block and ensure its veracity. This act of ensuring veracity is called *validating* or *importing* a block.

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:

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

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

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

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

[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

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 (18). And from 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 (19). Aside from

[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.

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, 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 (20). 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.

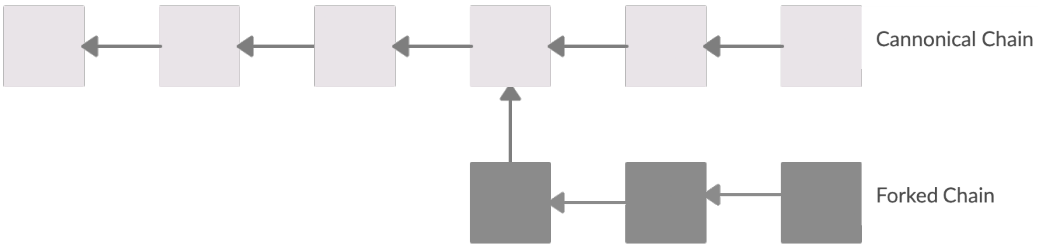

Figure 2.2: Forks - The cannon chain and the forked chain both have a common prefix, yet have *different* best-blocks.

add genesis to the figure

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

2.1.3.10 Merkle Tree and State 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 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] (21). 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

[1] Sometimes referred to as "Trie" as well.
[2] Named after Ralph Merkle, who also contributed to the foundation of cryptography in (8).

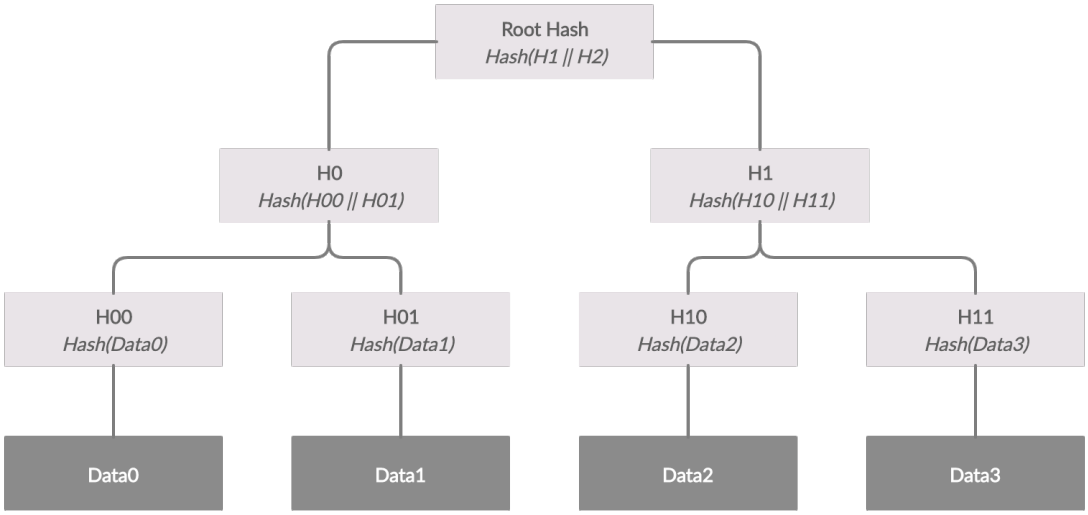

Figure 2.3: Merkle Tree - A binary Merkel Tree. The root hash contains a digest of all the 4 data nodes.

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 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).

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 to 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 a number of smaller groups, each of which can then be executed out of order (concurrently), without causing an invalid outcome.

The outcome of interest, of course, is only the aforementioned **state database** after all of the transactions of the block are applied. 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[1]. As mentioned, this comparison is done by the means of state root.

Do a pass and remove all of this as-mentioned crap.

.

Make the stype off all defenitions similar.

.

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.

[1]Note that this is only the case of block validation (block *import*). There is also the task of block authoring which is actually more complicated, but irrelevant to the discussion of this section – see 2.1.1.

This is in fact the property that ensures that the state is, in principle, optional to keep around, because it is deterministically reproducible 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 moot. The challenge is to allow these threads to cooperate and achieve an effective concurrency, while still maintaining determinism. Therefor, we have translated our blockchain scenario to 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*.

I could elaborate more on the def. of concurrency control but meh this is enough.

.

2.2.1 Locking: Pessimistic Concurrency Control

Locks are a common and intuitive[2] way for concurrency control. A lock is an abstract marker 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 (22, 23).

The simplest form of a lock will not distinguish between reads and writes and the only operation on it are `acquire` and `release`. To access the data protected by the lock, first the lock itself needs to be acquired. 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

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

[2] Sadly, as we will see the most intuitive way is not always the easiest to use.

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 (24).

A more elaborate variant of Mutex is read-write locks (RW locks). Such locks leverage the fact that multiple reads from the same data are almost always harmless and should be allowed. Thus, the read and write distinction. 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 programming 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 (25).

Locks are easy to understand, but notoriously hard to use. A programmer needs to delicately think about every single critical memory access and acquire and release locks from different threads to prevent wrong outcomes. Even worse, immature use of locks often leads to programs having deadlocks, the state in which all threads are waiting for a lock acquired by another thread. These issues are common programming errors and very hard to detect and resolve.

ref for this par.

Moreover, locks are a pessimistic means of concurrency control. They assume if two threads want to acquire the same (write) 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. This is exactly what the next section will address.

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" (26).

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 are conflicting or not, 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. The great advantage of this model is that if two transactions access the same memory region, but in a non-conflicting way[1], then there is a lot less waiting. In essence, there is *no* waiting in the execution of transactions, at the cost of some runtime overhead when they want to commit.

Transactional memory can exist either via a specialized hardware or, simulated in the software, referred to as *software* transactional memory, or STM (27). If implemented in the software, it does incur a runtime overhead. Nonetheless its programming interface is much easier and less error prone, because the programmer need not to manually acquire and release locks. (28).

Transactional memory is likely to lesson the waiting time and conflicts. Nonetheless, it still allows threads to operate over a same data structure. This implies complications about commits, aborts, and coherence[2]. 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 workload 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, 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,

[1] Textbook example: Access two different keys of a concurrent hash map.

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

and it just the matter of which will do it first. Again, both outputs are correct, yet the program is not deterministic.

Remark. A reader can at this point link our use of words "correct" and "determinism" to 2.1.3.7 and block authoring specifically: 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). All such blocks are probably *correct*. The first and foremost importance is for all validators to *deterministically* 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." (29). 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 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 (TODO again cite).

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 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 static analysis is to somehow improve the concurrency degree, by leveraging only compile-time information. In the case of transactional memory, this could be achieved by using static analysis to predict and reduce aborts(30). Similarly, other studies have tried to use static analysis to improve the usage of locking by

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

automatically inserting the lock commands into the program's source code at compile time (31). This can greatly ease the user experience of programmers using locks and reduce the chance of human errors possibility.

Similar to message passing, we inspire from the underlying concept of static analysis in our design later in chapter 3.

This brings our brief introduction to concurrency to an end. Next, we will use this entire chapter to formulate our approach to concurrency. https://www.overleaf.com/project/5ea596da15b938000163

3

Approaches toward Concurrency

In this chapter we will build up all the details and arguments needed to introduce our approach toward concurrency within blockchains. We will start with an interlude, enumerating different ways to make blockchains achieve a higher throughput from an end-to-end perspective, at the end of which we point out concurrency as our method of choice.

3.1 Prelude: Speeding up a Blockchain - An Out of The Box Overview

As mentioned, blockchains can be seen, in a very broad way, 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. 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 throughput [1].

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. As an example, two common consensus protocols are Proof of **Work** and Proof of **Stake**. They use the computation power (*work*) and an amount 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* different throughput characteristics (32). Proof of work, as the names suggests, requires the author to prove their legitimacy by providing a proof that they have

[1] and this categorization is by no means exhaustive. We are naming only a handful.

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 frequency of blocks, which directly impacts 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 (2).

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[1], a smaller block-time is not insecure in itself. Recently, we are seeing blockchains turning into verifiable random functions (33) for block authoring, and deploy a traditional byzantine fault tolerance voting scheme on top of it to ensure finality (34, 35). This further *decouples block production and finality*, allowing production to proceed faster and and with even less drag from the rest of the consensus system, namely finality.

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, specifically it which 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 where private blockchains were named as being always the fastest.

3.1.1.1 Sharding

An interesting consensus-related optimisation that is gaining a lot of relevance in recent years 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 that 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 sub-chains all have the same prefix and only differ in the most recent blocks. At fixed intervals, sub-networks come to agreement and synchronize their shards with one another. In some sense, sharding allows smaller sub-networks to progress faster(36, 37, 38).

[1] Using some hypothetical election algorithm which is irrelevant to this work.

3.1.2 Chain Topology

Another approach is changing the nature of the chain itself. A classic blockchain is theoretically limited due to its shape: a chain has only one head, thus only one new block can be added at each point in 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 (or individual transactions) to be created at the same time. Consequently, this approach turns a blockchain from a literal *chain of blocks* into a *graph of transactions* (39). Most often, such technologies are referred to Directed Acyclic Graphs (DAG) solutions. A prominent example of this is the IOTA project(40).

Allowing the chain to grow from different heads (i.e. seeing it as a graph) allows true parallelism at the transaction layer, effectively increasing the throughput. Nonetheless, the security of such approaches is still an active area of research and achieving decentralization with such loose authoring constraints has proven to be challenging (41).

Altering chain topology will bring even more radical changes to the original idea of blockchain. While being very promising for some fields such as massively large user applications (i.e. "Internet of Things", micro-payments), we will not consider DAGs in this work. We choose to adhere to the definitions provided in chapter 2 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 any chain-specific detail. Any chain will eventually come to a point where it must execute some transactions, be it in the form of a chain, or a DAG, with any consensus. Thus, concurrency is a viable as long as notion of transactions and blocks exists.

Our work specifically focuses on this aspect of of 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 Summary

I am not yet super happy with this section. I think I can write it in a more compact + confident way

.

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 throughput, not aiming for a specific absolute rate. It is true that in a specific chain, the network latency or consensus process might be the main bottleneck of the *overall 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, this does not mean that there is no merit to it. At any of these layers, any speedup is valuable and translates to more efficient use of the hardware.

Therefore, we argue that our method is *valuable* and *applicable*, regardless of being the dominant factor or not. For some chains, it might be a dominant factor and drastically improve the throughput, whilst in others it might not be and only allow the system to better use the hardware that is provided to it[1].

Finally, by seeing these broad options, we can clarify our usage of the word "throughput". One might notice that the first two options mentioned in this section (consensus, chain topology) can increase the throughput at the *block* level: More blocks can be added, thus more transaction throughput. This is in contrast to what concurrency can do. The concurrency explained in 3.1.3 is the matter of what happens *within* ***a*** block. Henceforth, by throughput we mean throughput of transactions that are being executed within a (*single*) block. Similarly, by concurrent, we mean concurrent within the transactions of a (*single*) block, not concurrency within the blocks themselves.

3.2 Concurrency Within Block Production and Validation

In this section, we will explain in detail how a concurrent blockchain will function. Most notably, we define how a concurrent author and a concurrent validator differ from their sequential counterparts. Note that these are the mandatory requirements that *any* approach toward concurrency in blockchains must respect. As the reader might expect based on previous explanations, all of them boil down to one radical property: **determinism**.

To recap, the block author is the elected entity that proposes a new block consisting of transactions. The block author must have already executed these transaction in some protocol-specific order (e.g. sequential), and note the correct state root of the block in

[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.

its header. This block is then propagated over the network. All other nodes validate this block and if they all come to the same state root, they append it to their local chain. An author that successfully creates a block gets rewarded for their work by the system.

3.2.1 Concurrent Author

We will begin by the chronologically sensible way, block authoring. Before anything interesting can happen in a blockchain, someone has to author and propose a new block. Else, no state transition happens.

A concurrent author has access to a pool of transactions that have been received over the network, most often via the gossip protocol. Form a consensus point of view, it is absolutely irrelevant to ensure all nodes have the same transactions in their local pool. In other words, form a consensus point of view, there is no consensus in the transaction pool layer. All that matters is that any node, once chosen to be the author[1], has a pool from which it can choose transactions. Then, the author has a *limited* amount of time to prepare the block and propagate it.

A number of ambiguities arise here. We will dismiss them all to be able to only focus on the concurrency aspect.

- Typically, the author need some way to *prefer* a subset of the transactions pool, as most often all of it cannot be fit into the block. For this work, we leave this detail generic and assume that each author has first filtered out its *pool* into a new *queue* of transactions (noting that the former is *unordered* and the latter is *ordered*) that she prefers to include in the block. In reality, a common strategy here is to prioritize the transactions that will pay off the most fee, as this will benefit the block author.

- A block must have some chain-specific *resource* limit. For simplicity,we assume that each block can fit a fixed *maximum* number of transactions, but it is so high that the bottleneck is not the transaction limit itself, but rather the amount of *time* that tha author has to prepare the block. Thus, bolstering the importance of high throughput. In reality, some blockchains have adopted a similar approach (cap on *number* of transactions), or limit the size of the block (cap on encoded *byte length* of the transaction). Complex chains that support arbitrary code execution will even go further and limit the *computation* cost of the transactions, such as Ethereum's gas metering(42).

[1] The decision of how the author is elected is within the details of the consensus protocol.

Having all these parameters fixed, we can then focus on the block building part, namely executing each transaction and placing it in the block. A sequential author would simply execute all the transactions one by one (in some order of preference) up until the time limit, and calculate the new state root. These transactions are then structured as a block. Concatenated with a header that notes the state root, the block is ready to be propagated. The created block is an *ordered* container for transactions, therefore it can be re-executed deterministically trivially by validators, as long as everyone does it sequentially.

A concurrent author's goal is to execute these transactions in a concurrent way, hoping to fit *more* of them in the same limited *time*, while still allowing the validators to come to the same state root. This is challenging because most often concurrency is non-deterministic. Therefore, the author is expected to piggy-back some auxillary information to its block that allows validators to execute it deterministically. maintaining determinism is the first and foremost criteria of the concurrent author.

Moreover, the secondary criteria is a net positive gain in throughput. The concurrent author prefers to be able to execute more transactions within a fixed time frame that she has for authoring, for she will then be rewarded with more transaction fees.

3.2.2 Concurrent Validator

A validator's role is simpler in both the sequential and concurrent fashion. Recall that the a block is an ordered container for transactions. Then, the sequential validator has a trivial role: re-execute the transactions sequentially and compare state roots. The concurrent validator however, is likely to need to do more.

More specifically, the concurrent validator knows that a concurrent author must have executed all or some of the transactions within the block concurrently. Therefore, conflicting transactions must have **proceeded** one another in *some* way. The goal of the concurrent validator is to re-produce the *same precedence* in an efficient manner and come to the same state root.

For example, Assume the author uses a simple Mutex lock to perform concurrency. In this case, some transactions will inevitably have to wait for other transactions that accessed the same mutex earlier. This implies an *implicit* precedence between conflicting transactions. The author will need to somehow transfer these precedence information to the validator, and the validator must respect them in order to arrive at the same state root.

With this background, we will briefly survey existing approaches in the literature to achieve concurrency, effectively seeing some a more practical perspective of the above description.

3.3 Existing Approaches

I am a bit tempted: Do I even need a related work section after all?

In this section we will look at some of the already existing approaches toward concurrency in blockchains. While doing so, we denote their deficiencies and build upon them to introduce our approach.

3.3.1 A Closer Look

Every tool that we named for concurrency in chapter 2 can essentially be used in blockchains as well, yet each will have a specific toll on the system in order to be feasible. All of these approach fall within the category of **concurrency control**. We will begin by arguably the simplest, locking.

A locking approach would divide the transactions into multiple threads[1]. Each transaction within the thread, when attempting to access any key in the state[2], has to acquire a lock for it. Once acquired, the transaction can access the key. As mentioned this process is not deterministic. Therefore, the runtime need to keep track of which locks were requested by which thread, and the *order* in which they were granted. This information is, in essence, builds a dependency graph. This dependency graph need to be sent to the validators as well. The validators parse the dependency graph and based on that spawn the required number of threads, and distribute the transactions within them. (43) is among the earliest works on concurrency within blockchain and adopts such an approach.

The details of generating the dependency graph with minimum size, encoding it in the block in an efficient way and parsing it in the validator is being highly simplified here. These steps are critical, as they are the main overheads of this approach. The size of this graph need to be small, as it needs to be added to the block and increases the network overhead. Moreover, the overhead of this extra processing must be worthwhile for the author, as otherwise it would be in contrast to the whole objective of deploying concurrency. There are some works that only focus on the "dependency graph generating and processing" aspect of the process. They assume some means exists through which the read and write

[1] a 1:1 relation between threads and transactions is also possible, given that the programming language supports green threads.

[2] Recall that the state is a key value database.

set of each transaction can be computed (i.e. by monitoring the lock requests that each thread sends at runtime). On top of this, the provide efficient ways to build the dependency graph, and use it at the validator's end (44).

The next step of this progression is to utilize transactional memory. The line of research exactly follows the same pattern. More recent works use software transactional memory to reduce the waiting time and conflict rates. Similar to the locking approach, the runtime needs to keep track of the dependencies and build a graph that encode this information. Different flavours of STMs are used and compared in this line of research, such as Read-Write STMs, Single-Version Object-based STMs, and Multi-Version Object-based STMs (45, 46). Nonetheless, the underlying procedure stays the same: Some means of concurrency control to handle conflicts, track dependency and use it to encode precedence, then re-crete the same precedence in the validator.

Next, we name an out-of-the box work that take a rather different path. Many studies in the blockchain literature use datasets from database industry as their reference. Such datasets might have unrealistically high rates of contention. (47) is an empirical study that tries to determine the conflict rates within Ethereum transactions, a *live*, and arguably well adopted network. And, while doing so, it demonstrate a different, wait-free approach. In the concurrent simulator of this work, all transaction are executed in parallel with the assumption that they will not conflict. If a conflict happens and a transaction aborts, it is discarded, and re-executed again at later phase sequentially. This essentially clusters transactions into two groups: concurrent and sequential. All of the concurrent transactions are guaranteed not to conflict. The sequential transaction do not matter as they are executed sequentially. Aside from their findings about the conflict rates in different periods of time in Ethereum, they also report speedups in *some* cases, not being too shy of the speedup amounts reported by (43), which uses locking.

This is an inspiring finding, implying that perhaps complicated concurrency control might not be needed after all for many of the transactions in *some* periods of time, based on the contention of the transactions. In some sense, this work adopts a technique that we coin as **concurrency avoidance** instead of **concurrency control**. As a consequence, the system need not to deal with conflicts in any way, because they are rejected and dealt with separately in the sequential phase.

Finally, we note that there has also been *some* work on pure static analysis in the field of blockchains, yet all of those that we have found require fundamental changes to the programmable language of the target chain. For example, (48) provides an extension to the Ethereum's smart contract language, Solidity, that allow it to be executed in a truly

concurrent manner (by essentially limiting the features of the language). Similarly, RChain is an industrial example of a chain that has a programming model that is fundamentally concurrent(49), namely pi-calculus(50). Such approaches are also inspiring, yet we prefer devising an approach which does *not* need to alter such fundamental assumptions about the programming model, in favour of easy adoption and outreach.

This is basically the answer to RQ1

3.3.1.1 Smart Contract vs. Runtime

Most of the mentioned references name their work as appraoches toward concurrency for **smart contracts**. At this point, it would be helpful to clarify that. The details of smart contract chains as well beyond the scope of this work. But, it is worth noting that a smart contract chain is a fixed chain that a fixed state transition logic that, and a part of that logic is to store codes (smart contracts) and execute them upon being dispatched. Moreover, since Ethereum is the prominent smart contract chain, all of these works present themselves with simulators that can hypothetically be implemented in the Ethereum node. To the contrary, we won't limit ourself to smart contracts or any specific chain in this work and build upon the basis that the future of blockchains will not be a *single* chain (chain maximalism), but rather an abundance of domain specific chains interoperating with one another. To achieve this, one needs to think in the context of a framework for building blockchains, not a particular blockchain per se. Thus, we argue that our approach is applicable to any chain with independent runtimes, not any particular smart contract being executed in the runtime of another chain.

3.3.2 Conclusion: Finding a New Balance

All of the works mentioned above report positive speedups. Yet, we argue from a different, analytical and critical perspective and point out some overheads associated with them that can be improved, or challenges that can be avoided altogether.

Both locking and transactional memory will result in a sizeable overhead while authoring. This is mostly hidden in some sort of runtime overhead, for example the need to keep track of locking order, and consequently parsing it into a dependency graph. Moreover, this will inevitably increase the block size, because the dependency graph need to be propagated to all other nodes. Finally, the validator also needs to tolerate the overhead of parsing the dependency graph and making informed, potentially complicated decisions based upon it.

These are all overheads compared to the basic sequential model. In essence, we express skepticism toward these complex runtime machinery to deal with conflicts, and record

precedence. On the other hand the pure *concurrency avoidance* model is likely to fail under any workload with some non-negligible degree of contention, because it basically falls back to the sequential model where most transactions are aborted and moved to the sequential model. Results from (47) show the same trend.

In our approach, introduced in the next section, we try to minimize these overheads by finding a new balance between the "concurrency control" and "concurrency avoidance" model. Moreover, we apply the same analogy between "runtime" against "static". While *some* runtime apparatus is needed to orchestrate the execution and prevent chaos, tracking all dependencies is likely to be too much. Similarly, while a purely static approach toward concurrency is a radical change to well-known programming models, we claim that *some* static hints could nudge the runtime by providing useful information.

—-

3.4 Our Approach: Almost Lock-Free, Delegation and Pseudo Static

In this section, we will describe our approach toward concurrency in a blockchain runtime, both in the authoring phase and in validation. This approach is based on three key pivotal ideas, explained in the next section.

3.4.1 Key Ideas

The key ideas of our approaches can be enumerated as follows:

3.4.1.1 Almost Lock-Free: "Taintable" State

We have already seen that locking is a common primitive to achieve shared state concurrency. In our approach, we relax this primitive such that any access to a shared state by a thread **does not incur long waiting times**, but instead, it might simply fail. To do so, we link each key in the state database with a taint value. If a key has never been accessed before, it is untainted. Once it is accessed by any thread (regardless of the type of operation being read or write), it is tainted by the identifier of that accessor thread. Henceforth, any access to this key by any other thread fails, returning the identifier of the original tainter (aka. *owner*) of the key. As we will see in the implementation details, this approach is *almost* wait free, meaning that threads will almost always proceed immediately with any state operation. Indeed, a thread can always freely access keys that it has already tainted before.

This is a good discussion section: we don't we distinguish them?

3.4.1.2 Concurrency Delegation

If a thread, in the process of executing a transaction, tries to access a tainted state key, then it forwards this transaction to the owner of that key. By doing so, a thread basically *delegates* the task of executing a transaction concurrently to another thread, namely because it cannot meet the state-access requirements of the transactions itself, and based on the available error information, the recipient (i.e. the owner of the failing state key) is more likely to be capable of doing that. This is the middle ground between *concurrency avoidance* and *concurrency control*, which we have coined *concurrency delegation*.

Compared to concurrency control, threads do not try to resolve contention in any sophisticated way. Instead, they will simply delegate (aka. *forward*) the transactions to whoever they think *might* be able to execute it successfully. There is no waiting involved, and no record being kept about access precedence.

On the other hand, comparing this with concurrency avoidance model, just because a transaction's state operation has failed, it does not mean that it cannot be executed in any concurrent way. Instead, it might be executable by another thread who is known to be the owner of the problematic state key. Therefore, instead of immediately being discarded (and potentially executed sequentially at the end), each transaction is given a second chance to succeed in a concurrent fashion.

Finally, if a transaction is already forwarded and still cannot be executed, (only) then it is forwarded to a sequential queue to be dealt with later. We name such transactions **orphan**. This implies that each transaction is at most forwarded twice: once to another potential recipient and lastly to the be declared as orphan.

Lemma 3.4.1. Detection of the Orphan Transaction.

Why "a failing transaction that has already been forwarded once" is considered an orphan? We present an example to demonstrate this: Assume thread $T2$ receives a forwarded transaction from $T1$. Based on the the protocol of delegation, we know that this transaction must have at least one key which is owned by $T2$, because $T1$ failed to access it and thus forwarded it to $T2$. Moreover, if the transaction still fails, it means that it has at least one other key that is owned by some other thread T'. This implies that $T2$ is not able to execute this transactions, and any further recipient of this transaction, including T', will also fail to execute it because the transaction has at least one key owned by $T2$. Therefore, no transaction will ever need to be forwarded more than once. A transaction can be forwarded at most once, and thereafter it is considered to be an orphan.

3.4.1.3 Pseudo-Static

We predict that concurrency delegations works well when threads need to seldom forward transactions to one another. In other words, the transactions need to be *initially* dis-

tributed between the threads in some educated and effective way to minimize forwarding. This is where we turn to pseudo-static heuristics as a form of a hint. We use the term pseudo-static because because we do not mean information that is necessarily available at compile time, but rather are known *before* a particular transactions is executed. In other words, they are *static with respect to the lifetime of a transaction* and can be inferred without the need to *execute* a transaction, but rather by *inspecting* it.

Our idea is based upon these key ideas. In the next section, we will connect the ideas together and depict how they will work together in the form of a unified algorithm.

3.4.2 Baseline Algorithm

Remark. For ease of understanding, we assume a system with 1 main (master) thread and 4 worker threads in this section. Needless to say, all of the details are applicable to systems with more or less degrees of parallelism.

We will first look into the baseline algorithm, which is essentially the concurrency delegation combined model and the taintable state, without any static heuristics. We assume a single master thread (henceforth called "master"), and 4 worker threads (henceforth called "worker"), each having the ability send messages over channels to one another. The master has access to a potentially infinitely large queue of (*ordered*[1], ready to execute) transactions. Moreover, it has an (initially) empty queue of orphan transactions, to which workers can forward transaction. Moreover, each worker has a local queue, to which the master and other workers can send transactions. The master and all workers share a reference to the same state database, S, which follows the logic of a taintable state as explained in 3.4.1.1.

Remark. As we will see, it is crucial to remember that both the transaction queue and the block are **ordered** containers for transactions. In other words, they both act like an ordered list/array/vector of transactions. The order of transaction in the queue will end up being used to order the transaction in the final block as well.

The master's (simplified) execution logic during **authoring** is as follows:

1. **Distribution phase**: The master starts distributing transactions between workers by some arbitrary function F. In essence, F is a $fn(transaction) \rightarrow identifier$, meaning that for each transaction, it outputs one thread identifier. Once the distribution is done, each transaction in the queue is *tagged* by the identifier if one worker thread. The distribution phase ends with the master sending each transaction to its corresponding worker's local queue.

[1] Recall that each node has an unordered pool of transaction, from which they chose an ordered queue based on some arbitrary preference

2. **Collection phase**: The master will then wait for reports from all workers, indicting that they are done with executing all of the transactions that they have received earlier. During this phase, threads might forward transactions to one another, and might forward transactions back to the master, if deemed to be orphan, exactly as explained in the concurrency delegation model. Both of these events are reported to the master and the tag of each transaction might change in the initial queue. Once termination is detected by the main thread, a message is sent to all workers to shut them down.

3. **Orphan phase**: Once all worker threads are done, the master will execute any transactions that it have received in its orphan queue. At this point, the master thread is sure that there are no other active threads in the system, thus accessing S without worrying about the taint is safe. The transactions are executed sequentially on top of S, and their tag is changed to a special identifier for orphan transactions.

Then, the master is ready to finalize the block. By this point, each transaction is either tagged to be orphan, or with the identifier of one of the 4 worker threads. In essence, we have clustered transactions into 4+1 *ordered* groups. The validator who receives this block will respect this clustering and execute transactions with the same tag in the same thread. In the first 4 groups, the transactions within a group might conflict with one another (e.g. attempt to write to the same key), but they are ordered and are known the be executed by the same thread, thus deterministic. The transactions in the last group, namely the orphan group, are executed sequentially and in isolation, thus safe.

In the same time frame, the worker thread will do as such (with slight simplifications):

1. **Depleting local queue**: Having received a number of transactions from the master (after the "*Distribution phase*"), each worker will then try to deplete its local queue. For each transaction Tx, the logic is as follows:

   If Tx is executed successfully, nothing is done or reported. This is because the master is already *assuming* that Tx *will be executed* by the current worker , therefore nothing need to be reported. If Tx fails due to a taint error, it is forward to the owner of the state keys that caused the failure. Note that at this point we know that Tx has not been forwarded before, because it is being retrieved from the initial local queue.

   At the end of this phase, the worker will send an overall report to the master, noting how many of the transactions it could execute successfully, and how many ended up being forwarded. This data is then used at the master to detect termination.

2. **Termination phase**: Once done with their local queue, the threads will listen for two messages, namely *termination* or *forwarded* transactions from other threads. Termination is the message from the master to shut down the worker. Forwarded transactions are those that another worker thread is *delegating/forwarding* to the current thread, namely because of a taint error. The transaction is then executed locally and if it is successful, the result is reported to the master. If the execution fails again due to a taint error, then it is forwarded to the master as an orphan.

   Note that in this case, reporting is vital, because a thread is ending up executing a transaction and the master is *not* aware of this, because it is not the same tag assigned in the "Distribution phase" of the master. This is also needed for the termination detection of this phase.

3.4.2.1 Analysis and Comments on the Baseline Algorithm

A number of noteworthy remarks exists on the baseline algorithm:

Termination Detection. We intentionally did not describe how the master detects the termination of the collection phase, because in doing so we needed further information from the worker thread's logic. Recall that the master knows how many transactions it has initially distributed between all the workers. Moreover, from the reports sent by thw worker at the end of "Depleting local queue" phase, it knows how many of them executed in their *designated thread*, and how many of them ended up being *forwarded*. Also, recall that each forwarded transaction, upon being executed successfully, is reported to the master. Similarly, each forwarded transaction that fails is also reported to the master (by being forwarded to the orphan queue residing in master thread). Thus, the master can safely assert that: Termination is achieved once the sum of "all locally executed transactions at workers", "forwarded and successfully executed transactions" and "orphan transactions" is equal to the initial count of transactions in the queue. At this point, the termination message is created and broadcasted to all workers.

Definition 3.4.1. Termination of the main thread's collection phase.
Assume N initial transaction. Each worker thread, upon finishing the "deplete local queue" phase reports back $\{l_1, l_2, l_3, l_4\}$, indicating the number of transactions that a worker executed locally. Given F being the number of reports of transactions being forwarded, and O being the size of the orphan queue, termination is achieved once

$$N == \sum_{t=1}^{4} l_t + F + O \tag{3.1}$$

Maintaining Order: Aside from termination detection, it is also vital for determinism that the master takes action upon the report of a transaction being forwarded. This is because **once the tag of a transaction changes, it is likely that its order must also change within the queue**. For example, if a transaction, initially assigned to T_1 is known to be forwarded and executed by T_2, it is important to re-order it in the initial transaction queue such that it is placed *after* all the transactions initially assigned to T_2. This is because, in reality, T_2 *first* executed all of its designated transactions and *then* executes any forwarded transactions. Recall that the queue is an ordered container for transactions and its order will eventually end up building the order of transaction in the block.

Orphan Transactions. Recall that an orphan transaction is a transaction has already been forwarded and still fails to execute, at its current host thread, due to taint error. We now represent this from a different perspective. In our concurrency delegation scheme, threads race to access state keys and upon successful access, they taint them. Any transaction has a number of state keys that it needs to access in order to be processed. **An orphan transaction is one that has state keys being tainted by at least two *different* threads**. For example, assume a transaction needs to access keys K_1 and K_2. Assume thread T_0 is executing this transaction. If K_1 is already tainted by a T_1 and K_2 by a T_2, then this transaction will inevitably end up being in the orphan queue. The transaction is first forwarded from T_0 to T_1, where it can successfully access K_1, but still fails at accessing K_2 and thus orphaned[1].

Minimal Overhead. As we have seen, our approach incurs minimal overhead to the block. In fact, the only additional data needed is one identifier that need to be attached to each transaction, indicating which thread must execute it (and the special case thereof, orphan transaction), essentially the *tag*. This can be as small is a single byte per transaction, which is not much. Note that transactions within the block still maintain partial order: The transactions of a particular tag are sorted within their tag. Only the relative order of transactions from different tags is lost, which is not significant, because they are guaranteed by the author to not conflict.

Validation. We can see now consider validation as well. As expected, due to the minimal overhead and the simplicity of the baseline algorithm, the validation logic is fairly simple. Each block is received with all of its transactions having a tag. The ones tagged to be orphan are set aside for later execution. Then, one worker thread is spawned per tag and transactions are assigned to threads based on their tags, and in the same relative order.

[1] An interesting optimization can be applied on top of this logic, which is explained further in TODO.

The workers can then execute concurrently, without the need for any concurrency control, because they effectively know that all contentious transactions already have the same tags, and thus are ordered *sequentially within that tag/thread*. In essence, the validation is fully *parallelizable* and does not need any synchronization. Once all threads are finished, the orphan transactions are executed sequentially and validation comes to an end.

3.4.2.2 Proof: Determinism in Concurrency Delegration

Finally, we must address the most important criteria, **Determinism**. We provide the proof by showing that the validation and authoring both have the exact same execution environment, and the transactions are executed in the exact same order in both phases. In more detail, both the author and the validator will spawn the same number of threads, and all of the transaction executed by any thread during authoring are re-executed by a single thread in the validation phase as well.

First, consider all transactions that are executed in their initially designated thread, based on the aforementioned distributor function F. All of these transactions are assigned a tag, and have some partial order within the tag. The important note is, that they are also placed in their designated worker thread's queue with the same order, and thus executed in the same order. Consequently, they are, yet again, placed in the final block with the same order within the tag. Therefore, it is trivial to see that the worker thread in the authoring phase and the worker thread in the validation phase will execute *exactly the same* transactions, in the exact same order.

Next, consider the transactions that ended up being forwarded. These transactions ended up being executed *after* the designated transactions of their final host thread. The main thread, responsible for building the final block, have to note this change and ensure that this partial order is maintained in the final block. The first step for the master is to change the tag of this forwarded transaction to the tag of its newly designated thread. Then, to ensure that the order *within the tag* is maintained, master thread will simply place this transaction at the end of the queue. This will ensure that this transaction will be placed in the final block in such a way that it is executed after all the transactions that have the same tag. Then, we can apply the same logic as the previous paragraph and assert that the order of execution of all transactions with a specific tag stays the same within one thread in authoring and validation, thus deterministic.

Last but not least, the orphan transactions also need the same property to be deterministic: maintaining order. The master thread needs to make sure that all the transactions that are tagged as `Orphan` within the block have the same order within them as they were

executed. With the combination of these notes, we can assert that our approach is fully deterministic, with minimum effort. Indeed, the only subtlety that needs to ba taken care of is re-ordering of a transaction in the queue (and consequently the block) when it is *successfully* forwarded.

3.4.3 Applying Static Heuristics

The previous section is a complete description of a system that is deterministic and can be deployed as-is, without any further requirements. Nonetheless, one can argue that this system might not be optimal in the throughput gain that it can deliver. This is because we not said anything about the distribution function of the master, namely F. While we keep this distribution function generic in this entire work, we will make one important claim about it, namely that using static information of the transaction can be very beneficial and is key to an optimal performance.

A static information is anything that can be known about a transaction, before it hits the runtime and gets executed. In other words, we are interested in any information that can be inferred from the transaction, *without* the need to *execute* it. A clear example of this is the origin of the transaction (i.e. the sender account). Because of public key cryptography usage, all transactions carry a signature and the origin account as a part of their payload. Therefore, using the origin as a static information is permitted, because it is known even *before* the transaction is executed. The similar analogy applies to the arguments of the transaction as well. These are information that are encoded in the payload of the transaction and the runtime of the master thread (before starting authoring) can effectively use them to optimise F. To the contrary, consider the return value of the transaction. This is a piece of information that is not considered static with respect to the transaction, because it can only be known by executing the transaction.

Remark. Given the above paragraph, we denote that out definition of static is different than the term which is usually referred to compile-time information. Therefore, we used the term pseudo-static in some places to delineate the difference.

We use this ability to our benefit by proposing a static annotation can be added by the programmer to each transaction. Recall that the underlying state of the blockchain runtime is a key-value database, so each state access is linked to a key. Moreover, if F can know the list of keys accessed by each transaction it could create a perfect distribution where no transaction is ever forwarded or orphaned, because no thread will ever reach a taint error while accessing the state. Of course, things are not simple. In practice, it is

impossible to know the execution path of a complex transaction without executing it[1], therefore knowing the exact state keys that it must access is also impossible.

The important point is to remember that the annotation can still be reasonably accurate, without the need for executing the transaction. This annotation can state, in a best effort manner, which state keys are *likely* to be accessed by this transaction, based on, for example, one of the common execution paths of the transaction.

The following listing shows an abstract example of this:

Listing 3.1: Example of Static Hints

```
#[access = (origin.state_key(), arg1.state_key())] fn transaction(origin, arg1,
    arg2) { /* ... */ }
```

We observe a transaction, embodied as a function named `transaction`, which has 3 arguments, the origin and two auxillary ones. Furthermore, we observe a macro[2] that is providing the state keys that *might* be accessed by this transaction. In this case, the second argument is not relevant and seemingly only some state key of the origin and the `arg1` might be accessed.

This macro syntax is just one example, nonetheless its specification is irrelevant at this point. What matters is: The transaction can provide the runtime with some easy-to-compute, pseudo-static information about the *state access* of the transaction (therefore we called the macro `#access` in the listing 3.1). The runtime can then effectively use this information in the transaction distribution phase to come up with a better distribution that leads to less contention, consequently less forwarding and less orphans.

Remark. Why do we don't execute in the pool and all? why hints must be cheap to compute? DOS.

3.4.4 System Architecture

We have essentially answered RQ2 by the end of this chapter.

In this section, we bring 3.4.2 and 3.4.3 into one picture, and explain the entire system architecture. We tackle this by presenting an all in one picture of the system architecture and explaining all the components, connecting all the dots as we move forward.

Figure 3.1 includes enumerated labels on each system component, and we use these labels to explain each component.

[1] An interesting reader can refer to the "Halting problem" for a formal representation of this issue (51).

[2] This style refers to the Rust programming language, but is applicable to any other compiled language as well.

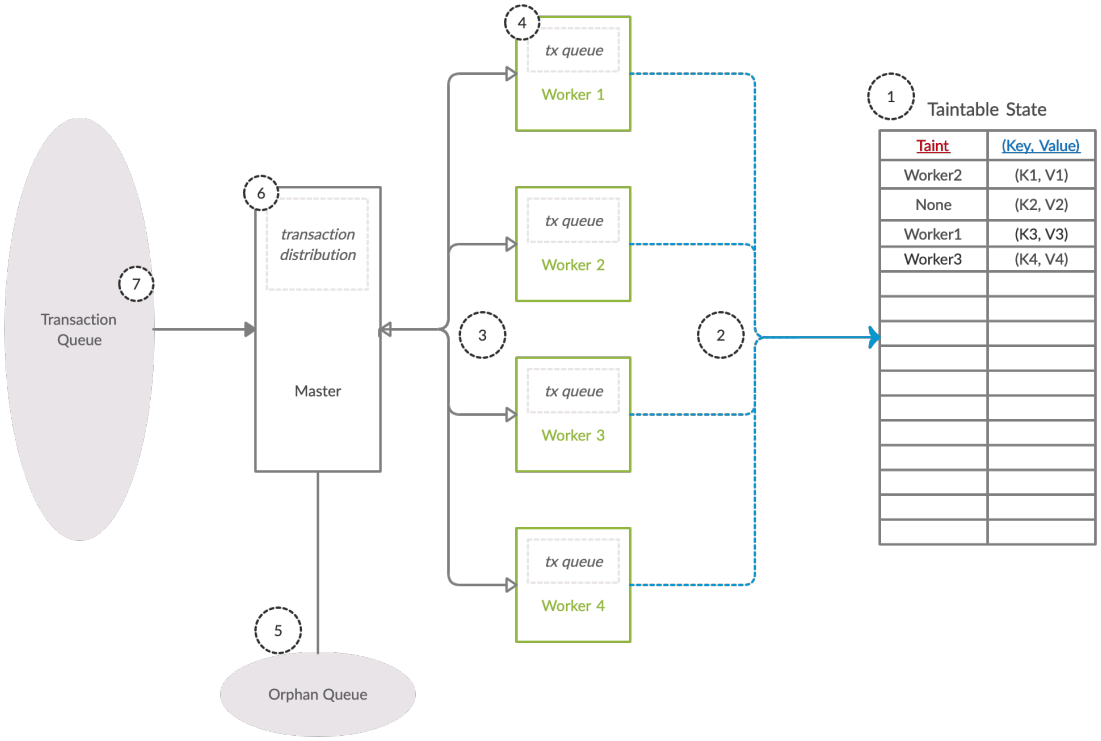

Figure 3.1: Overall System Architecture -

- **1** shows the taintable state. We can see that each cell is not merely a key-value pair, but rather a taint value, places next to a key value pair. Some keys are not tainted (`None`), while some are already tainted by a thread because they have been accessed by it. An access to each of the keys by a thread will succeed if the key is untainted, or is tainted by the accessor thread. Else, an error is returned, noting the identifier of the thread that owns the key.

- **2** is an emphasis on the access of all the workers to the state to bewait-free. In an abstract level, there is no waiting involved in this model. Any access to a key in the state will either *immediately* succeed, in which case the key is tainted. Else, it is *immediately* rejected because of a taint error.

unify the word lock f and wait free.

- **3** shows the communication channels between the master and workers. These channels allow all of the threads to communicate by the means of sending messages to one another. Most often these messages are simply a transaction to be forwarded. It is worth noting that worker threads can also communicate in the same manner.

- **4** shows the local queue of each thread. This is where the transactions that the

master thread designates to each thread live, until they are executed. Needless to say, this queue is also ordered.

- **5** shows the orphan queue, where the master thread maintains an ordered queue of transactions that are essentially rejected by the workers and need to be executed in a second sequential phase. Once all the workers are done accessing the state, the master thread will exclusively start using the state (essentially with ignoring all the taint values) and execute all the orphan transactions sequentially.
- **6** importantly depicts the transaction distributor component of the master thread. This component is invoked prior to the process of authoring to tag all the transactions that are ready to be executed in the transaction queue. This effectively determines which thread gets to execute which transactions.
- **7** is the transaction queue, where an ordered subset of the transaction pool is verified and awaiting to be delegated to worker threads.

With this overall blueprint of the system's architecture, we move forward to the next chapter, where we will bring this concept one step closer to reality by covering some of the implementation details.

4

Implementation

In this chapter we will bring the system architecture mentioned at the end of the chapter 3 closer to a real programming environment. This chapter is by no means extensive, since there are many many implementation details that could be worth noting. Nonetheless, in favour of brevity we will minimize the details to only those that are either of:

- Of importance with regards to the evaluation of the system.
- Impose a particular practical challenge that we find interesting.

For the implementation we use the Rust programming language (52). Being backed by Mozilla, Rust has a lot to offer in the domain of system programming and low level application, such as a blockchain client/engine. Rust is a unique language, among the few rare ones in which one can claim that the leaning curve is indeed steep and it is *not* just a matter of learning the new syntax. One of the reasons for this learning curve is Rust's compile-time memory management system, which means all allocations and de-allocations are inferred and checked at compile time. This ensures that the program is memory-safe, whilst having no garbage collector. Rust is, in some sense, the performance of C, combined abstractions and safety features of Java or C#(53).

Lastly, it is worth mentioning that our choice is not merely out of interest. Rust have been heavily invested in, in the last few years, by big blockchains companies and in the field of research(54).

4.1 The Rust Standard Library

First, we will explain some of the primitive types available in the Rust's standard library that we use. Note that these types are merely *our* choice for this implementation and similar data types from other libraries (or *crates*, in the Rust jargon) are also acceptable.

4.1.1 State: HashMap and Locks

For the taintable state, we need a data type that is similar to a typical concurrent `HashMap`(55), yet has slight differences. Rust doesn't even provide a concurrent HashMap in the standard library, so the only way to go is to implement our own custom data structure. To implement this, we use both a `HashMap` and a `RwLock`, both of which are provided by the standard library. The HashMap behaves just as a typical `HashMap` as in any programming language. The `RwLock` is a locking primitive with read-write distinction, where multiple *read* requests can be done at the same time, while a *write* request will block all access.

Like most data types in languages that support generics, the Rust's default `HashMap` is generic over both the key and value type that it uses. The final `HashMap` is using opaque byte arrays as both the key and value type. For the keys, we do our own hashing and concatenation to compute the location (i.e. the final key of) any given state variable. For example, to compute the key of where the balance of an account is stored, we compute: `"balances:balance_of".hash() + accountId.hash()` and use the final byte array as the key. As for the values, to be able to store values of different types in the same map, we encode all values to a binary format, thus, a byte array is used.

Listing 4.1: Key and Value Types.

```
1 /// The key type. type Key = Vec<u8>; /// The value type. type Value = Vec<u8>;
2
3 /// Final State layout. type State = HashMap<Key, Value>;
```

4.1.2 Threading Model

In short, Rust's threading model is 1:1. This means that each thread will be mapped to an operating system thread (which sometimes directly map to hardware threads). Naturally, this means that the overhead of spawning new threads is not negligible, but there are no additional runtime overheads. This justifies using a relatively small number of threads and assigning a large number of transactions to each, rather than, for example, creating one transaction per thread.

The main purpose of this decision is for Rust to remain a *runtime-free* programming language(56), meaning that there is zero-to-minimal runtime in the final binary. The opposite of a 1:1 threading model is a N:M model, also known as *green threads*. Such threads are lightweight, and do not map to a hardware/operating-system thread in any way. Instead, they are a software abstraction and are handled by a runtime. Therefore, A

language that wants to support green threads needs some sizeable runtime machinery to be able to handle that.

4.1.3 Communication

For communication, we use multi-producer-single-consumer(57) (`mpsc` for short) channels. Our choice for this type of channel is restricted by the fact that this is the only one available in the standard library and we preferred keeping the implementation dependency-free. This entails that, as opposed to the figure 3.1 that depicted a a shape that resembles a *unified bus* between all threads, the actual layout is rather a bit more complicated.

The process of using Rust `mpsc` channels is such that each channel has one receiver and one producer handle, ands the producer handle can be freely cloned into different threads (while the receiver cannot be copied around – this is ensured by the Rust's compile time checks.). With this approach, each thread will have a `mpsc` channel and keeps the receiving handle to itself, while giving a copy of the producer handle to all other threads.

The following listing demonstrates this process.

Listing 4.2: How Channles Allow Commication Between Threads (with slight simplification)

```
// In the local thread.
let (producer, receiver) = std::mpsc::channel();

// spawn a new thread. std::thread::spawn(|| {
  // this scope is local to a new thread. let
  local_producer = producer.clone();
  // note the cloned handle ----^^^^^^^

  local_producer.send("thread1");
});

// spawn another new thread.
std::thread::spawn(|| {
  // this scope is local to a new thread.
  let another_local_producer = producer.clone();
  // note the cloned handle -----------^^^^^^^

  another_local_producer.send("thread2");
});

// check incoming messages.
while let Ok(msg) = receiver.rcv() {
  // do something with `msg`.
}
```

4.2 Example Runtime: Balances

Next, we demonstrate an example runtime to help the reader familiarize themselves with the context of the implementation and how the final outcome looks like. Our final implementation supports having multiple *modules* within the blockchain runtime, each having their own specific business logic. The simplest example of such module is a balances modules that takes care of storing the balance of some accounts and allows transfer of tokens between them. We will now enumerate some of the important bits of code involved in this modules.

First, there needs to be a struct to store the balance of a single account, which we named `AccountBalance`.

Listing 4.3: Balance Strict

```
/// The amount of balance that a certain account.
#[derive(Debug, Clone, Default, Eq, PartialEq, Encode, Decode)]
pub struct AccountBalance {
  /// The amount that is free and allowed to be transferred out.
  free: Balance,
  /// The mount that is reserved, potentially because of the balance being used
      in other modules.
  reserved: Balance,
}
```

The support for reserved balances allows more sophisticated testing, further details of which are out side the scope of this chapter.

The state layout of this modules is simple: There needs to be one mapping, with the key being an account identifier and the value being `AccountBalance`, as in listing 4.3.

Listing 4.4: State Layout of the Balances Module

```
decl_storage_map!(
  // Auxillary name assigned to the storage struct.
  BalanceOf,
  // Auxillary name used in key hashing.
  "balance_of",
  // Key type.
  AccountId,
  // Value type.
  AccountBalance,
);
```

Note that the above statement is a macro (note the `!` notation), meaning that it generates a great amount of code at compile time. Most of this code deals with functions for generating the key of a specific account's balance, namely the hashing and concatenation

method explained in 4.1.1. Nonetheless, the inline documentation of the listing should be clear in delivering: There exists a state *mapping* from `AccountId` to `AccountBalance`".

We have already seen what `AccountBalance` is, and `AccountId` is an alias for –no surprise– a public key.

Finally, we can look at the only public transaction that can be executed in this module: a transfer of some tokens from one account to another one.

Listing 4.5: Transfer Transaction

```
1  #[access = (|origin| vec![<BalanceOf<R>>::key_for(origin),
       <BalanceOf<R>>::key_for(dest.clone())])]
2  fn transfer(runtime, origin, dest: AccountId, value: Balance) {
3    // read the balance of the
4    origin. let mut old_balance = BalanceOf::read(runtime, origin).or_forward()?;
5
6    if let Some(remaining) = old_balance.free.checked_sub(value) {
7      // origin has enough balance
8      old_balance.free = remaining;
9
10     // new balance of the origin.
11     BalanceOf::write(runtime, origin, old_balance).unwrap()
12
13     // new balance of the destination.
14     BalanceOf::mutate(runtime, dest, |old| old.free += value).or_orphan()?;
15
16     Ok(())
17   } else {
18     Err(DispatchError::LogicError("Does not have enough funds."))}
19   }
```

The logic of this transaction should be straightforward: Read the origin's balance. If they have enough free balance, update the balance of the origin and destination. Else, return error.

The more interesting bit is the lines 1-6, where our long promised `access` macro is being used in action. The interpretation of the macro is basically as follows: This transaction will (most likely[1]) access two state keys, the balance of the origin and the balance of the destination.

4.3 Generic Distributor

Another note-worthy detail of the implementation is the distributor component. Recall that the distributor is responsible for tagging each transaction in the transaction queue with the identifier of one thread. We emphasis that we leave this detail *generic* in our

[1] Recall that tha `access` macro was supposed to be a best effort guess.

implementation, similar to its position in the thesis. This means that there is no concrete implementation of a distributor embedded in our system. Instead, any function that satisfies a certain requirement can be plugged in and used as the distributor.

Two examples of distributor implementation as follows:

- **Round Robin**: This distributor will simply ignore the `access` macro and assign transactions to threads one at a time, in some random order.
- **Connected Components**(58): This graph processing algorithm is the exact opposite end of the spectrum compared to Round Robin, meaning that it *heavily* takes the `access` macro into account.

  The simplified explanation of it is as follows: All transactions provide a list of keys that they might access during their execution, via the `access` macro. This distributor will build a graph of transaction and state keys, where each transaction has an edge to all the keys that it might access. Therefore, two transactions that are likely to access the same state key will end up being *connected*, because they both have an edge to the same key. The connected component, as the name suggests, identifies these connected transactions. Once all components are identified, they are distributed among threads as evenly as possible. In essence, the connected component ensures that transactions that might conflict will end up being sent to the same thread, effectively minimizing forwarded and orphaned transactions.

4.4 Bonus: Taintable State

So far, all the mentioned details of this chapter are somewhat necessary to know in order to be able to comprehend the evaluation. Conversely, this last section is optional and explain some of the more details of the Taintable map implementation. Moreover, in this chapter we assume average Rust knowledge from the reader.

TODO

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, 30

[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] Stefano De Angelis, Leonardo Aniello, Roberto Baldoni, Federico Lombardi, Andrea Margheri, and Vladimiro Sassone. **PBFT vs Proof-of-Authority: Applying the CAP Theorem to Permissioned Blockchain**. In *Italian Conference on Cyber Security (06/02/18)*, January 2018. 16

[19] 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

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

[21] 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

[22] 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

[23] 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

[24] 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. 25

[25] Aaron Weiss, Olek Gierczak, Daniel Patterson, Nicholas D. Matsakis, and Amal Ahmed. **Oxide: The Essence of Rust**. *ArXiv190300982 Cs*, August 2020. 25

[26] 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

[27] 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. 26

[28] 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. 26

[29] **Share Memory By Communicating - The Go Blog**. https://blog.golang.org/codelab-share. 27

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

[31] Sigmund Cherem, Trishul Chilimbi, and Sumit Gulwani. **Inferring Locks for Atomic Sections**. August 2007. 28

[32] Alessio Meneghetti, Tommaso Parise, Massimiliano Sala, and Daniele Taufer. **A Survey on Efficient Parallelization of Blockchain-Based Smart Contracts**. *ArXiv190400731 Cs*, February 2019. 29

[33] Yevgeniy Dodis and Aleksandr Yampolskiy. **A Verifiable Random Function with Short Proofs and Keys**. In Serge Vaudenay, editor, *Public Key Cryptography - PKC 2005*, Lecture Notes in Computer Science, pages 416–431, Berlin, Heidelberg, 2005. Springer. 30

[34] Vitalik Buterin and Virgil Griffith. **Casper the Friendly Finality Gadget**. *ArXiv171009437 Cs*, January 2019. 30

[35] Alistair Stewart. **Poster: GRANDPA Finality Gadget**. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, pages 2649–2651, New York, NY, USA, November 2019. Association for Computing Machinery. 30

[36] Sébastien Forestier, Damir Vodenicarevic, and Adrien Laversanne-Finot. **Blockclique: Scaling Blockchains through Transaction Sharding in a Multithreaded Block Graph**. *ArXiv180309029 Cs*, September 2019. 30

[37] Mustafa Al-Bassam, Alberto Sonnino, Shehar Bano, Dave Hrycyszyn, and George Danezis. **Chainspace: A Sharded Smart Contracts Platform**. *ArXiv170803778 Cs*, August 2017. 30

[38] Baheti Shrey, Anjana Parwat Singh, Peri Sathya, and Simmhan Yogesh. **DiPETrans: A Framework for Distributed Parallel Execution of Transactions of Blocks in Blockchain**. *ArXiv190611721 Cs*, June 2019. 30

[39] Huma Pervez, Muhammad Muneeb, Muhammad Usama Irfan, and Irfan Ul Haq. **A Comparative Analysis of DAG-Based Blockchain Architectures**. In *2018 12th International Conference on Open Source Systems and Technologies (ICOSST)*, pages 27–34, December 2018. 31

[40] Divya M and Nagaveni B. Biradar. **IOTA-Next Generation Block Chain**. *Int. J. Eng. Comput. Sci.*, **7**(04):23823–23826, April 2018. 31

[41] Yonatan Sompolinsky, Yoad Lewenberg, and Aviv Zohar. **SPECTRE: A Fast and Scalable Cryptocurrency Protocol**. Technical Report 1159, 2016. 31

[42] Daniel Perez and Benjamin Livshits. **Broken Metre: Attacking Resource Metering in EVM**. *ArXiv190907220 Cs*, March 2020. 33

[43] Thomas Dickerson, Paul Gazzillo, Maurice Herlihy, and Eric Koskinen. **[SmartLocks] Adding Concurrency to Smart Contracts**. *ArXiv170204467 Cs*, February 2017. 35, 36

[44] *Enabling Concurrency on Smart Contracts Using Multiversion Ordering.* Springer Berlin Heidelberg, New York, NY, 2018. 36

[45] Parwat Singh Anjana, Sweta Kumari, Sathya Peri, and Archit Somani. **[STM] Efficient Concurrent Execution of Smart Contracts in Blockchains Using Object-Based Transactional Memory**. *ArXiv190400358 Cs*, August 2019. 36

[46] Parwat Singh Anjana, Sweta Kumari, Sathya Peri, Sachin Rathor, and Archit Somani. **[STM] An Efficient Framework for Optimistic Concurrent Execution of Smart Contracts**. *ArXiv180901326 Cs*, January 2019. 36

[47] Vikram Saraph and Maurice Herlihy. **[YOLO] An Empirical Study of Speculative Concurrency in Ethereum Smart Contracts**. *ArXiv190101376 Cs*, January 2019. 36, 38

[48] Massimo Bartoletti, Letterio Galletta, and Maurizio Murgia. **[Static] A True Concurrent Model of Smart Contracts Executions**. *ArXiv190504366 Cs*, May 2019. 36

[49] DARRYL. **RCast 21: The Currency of Concurrency**, March 2019. 37

[50] DAVID TURNER. **The Polymorphic Pi-Calculus: Theory and Implementation**. July 1996. 37

[51] L BURKHOLDER. **The Halting Problem**. *SIGACT News*, **18**(3):48–60, April 1987. 46

[52] STEVE KLABNIK AND CAROL NICHOLS. *The Rust Programming Language (Covers Rust 2018)*. No Starch Press, September 2019. 49

[53] RALF JUNG, JACQUES-HENRI JOURDAN, ROBBERT KREBBERS, AND DEREK DREYER. **RustBelt: Securing the Foundations of the Rust Programming Language**. *Proc. ACM Program. Lang.*, **2**(POPL):66:1–66:34, December 2017. 49

[54] **Rust in Blockchain**. https://rustinblockchain.org/. 49

[55] J. BARNAT, P. ROČKAI, V. ŠTILL, AND J. WEISER. **Fast, Dynamically-Sized Concurrent Hash Table**. In BERND FISCHER AND JACO GELDENHUYS, editors, *Model Checking Software*, Lecture Notes in Computer Science, pages 49–65, Cham, 2015. Springer International Publishing. 50

[56] **Rust's Journey to Async/Await**. https://www.infoq.com/presentations/rust-2019/. 50

[57] **Std::Sync::Mpsc - Rust**. https://doc.rust-lang.org/std/sync/mpsc/. 51

[58] ESKO NUUTILA AND ELJAS SOISALON-SOININEN. **On Finding the Strongly Connected Components in a Directed Graph**. *Information Processing Letters*, **49**(1):9–14, January 1994. 54