BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FAST-SYNC OF NEW NODES IN THE FANTOM OPERA PLATFORM

RYCHLÁ SYNCHRONIZACE NOVÝCH UZLŮ V PLATFORMĚ FANTOM OPERA

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR **Bc. MATĚJ MLEJNEK**
AUTOR PRÁCE

SUPERVISOR **Ing. MARTIN PEREŠÍNI**
VEDOUCÍ PRÁCE

BRNO 2022

Department of Intelligent Systems (DITS) Academic year 2021/2022

Master's Thesis Specification

25083

Student: **Mlejnek Matěj, Bc.**
Programme: Information Technology and Artificial Intelligence
Specialization: Cybersecurity
Title: **Fast Synchronization of New Nodes in the Fantom Opera Platform**
Category: Security
Assignment:

1. Get familiar with blockchain technology and existing protocols for fast synchronization of new nodes on different blockchain platforms (e.g., Bitcoin, Ethereum, Cardano, Fantom).
2. Study the current state and issues associated with new node synchronization, specifically in the Lachesis consensus within the Fantom Opera platform.
3. Analyze possible options for a fast-sync protocol in Fantom Opera and discuss their pros and cons.
4. Based on the analysis, design a fast-sync protocol with an emphasis on feasibility, performance, and security.
5. Implement the proposed protocol and evaluate it. Focus on the security and performance issues associated with it.
6. Discuss further improvements and limitations of your solution.

Recommended literature:

- https://fantom.foundation/fantom-research-papers/
- https://docs.fantom.foundation
- https://github.com/Fantom-foundation/go-lachesis/wiki
- https://arxiv.org/pdf/2108.01900.pdf
- https://geth.ethereum.org/docs/faq
- https://nipopows.com/

Requirements for the semestral defence:

- Items 1 to 3.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Perešíni Martin, Ing.**
Head of Department:	Hanáček Petr, doc. Dr. Ing.
Beginning of work:	November 1, 2021
Submission deadline:	May 18, 2022
Approval date:	November 3, 2021

Abstract

The goal of this master thesis is to research current ways of fast synchronization of new nodes in blockchain networks, considering different consenses. Then to compare these solutions, design feasible solution and then implement the new fast synchronisation for blockchain network Fantom Opera. The solution is based on an idea where new node in network doesn't connect to P2P network, but directly connects to one of the nodes and downloads just the necessary data to be synchronized with rest of network as fast as possible.

Abstrakt

Cílem této diplomové práce je prozkoumat aktuální způsoby rychlé synchronizace nových uzlů v rámci jednotlivých blockchain sítí dle závislostí na jednotlivých konsenzech, porovnat jednotlivá řešení mezi sebou, navrhnout možné řešení a naimplementovat novou rychlou synchronizaci pro blockchainovou síť Fantom Opera. Řešení spočívá v tom, že nově připojený uzel se místo do P2P sítě přímo připojí k existujícímu nodu a stáhne si od něj potřebná data pro co nejrychlejší synchronizování se zbytkem sítě.

Keywords

blockchain, synchronization, Fantom, Opera, Lachesis, Ethereum, PoS, Proof of Stake, encryption

Klíčová slova

blockchain, synchronizace, Fantom, Opera, Lachesis, Ethereum, PoS, Proof of Stake, šifrování

Reference

MLEJNEK, Matěj. *Fast-Sync of New Nodes in the Fantom Opera Platform.* Brno, 2022. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Martin Perešíni

Rozšířený abstrakt

Tato práce se zabývá způsoby sychronizace nových blockchainových uzlů do sítě. Nejprve jsem provedl analýzu existujících způsobů synchronizací na různých blockchainových platformách. Z nabraných poznatků jsem vytvořil řešení pro síť *Fantom Opera*. Opera (*go-opera*) je postavená na Ethereové Geth (*go-ethereum*) implementaci rovněž napsané v jazyku Golang a importuje z jejího repozitáře stále velkou část kódu. Hlavním rozdílem těchto dvou sítí je skutečnost, že Fantomová využívá Lachesis koncenzus, neboli způsob toho jak se o nových transakcí rozhoduje, zda budou zpracovány a potvrzeny je rozdílný. Lachesis je DAG, Proof of Stake a mezi jednotlivými uzly v síti se přeposílají události. Narozdíl Ethereum je Proof of Work a přeposílají se v něm transakces.

Moje první idea byla přenášet data přes P2P a inspirovat se implementací snapsync z Etherea, toto řešení ovšem bylo bohužel koncem minulého roku z velké části dokončeno[1]. Moje řešení se zaměřuje na synchronizaci nového uzlu tak, že místo toho, aby přenášela data z klasické P2P komunikace, tak se k síti nový server dosynchronizuje přes hostující server. Pokud by se použil přímý přenos zdrojových dat například pomocí nástroje *rsync*, tak bohužel nastane problém v tom že zdrojový server musí být pozastaven. Jelikož `rsync` prochází data v zdrojovém adresáři sekvenčně přímo z disku a při změně za průběhu přenosu by přenášená data v různých bodech databáze na sebe nenavazovala a takovýto nekonzistentní stav není možné opravit. Data jsem se rozhodl přenést iterací databáze po jednotlivých záznamech a za pomocí snapshotu - ten zajistí konzistenci díky jeho vlastnosti, že přenáší skutečný stav databáze přímo v době vytvoření onoho snapshotu. Díky tomu, že nové záznamy mají vyšší sekvenční číslo zápisu, je vytvoření tohoto snapshotu jako jednoduché zarážky prakticky instantní. Snapshot je vždy poznamenáván na konci epochy. Díky tomu není potřeba v databázi mít události z aktuálně prováděné epochy, toto je vlastnost DAG sítí, kde nové eventy jsou odkazovány na hlavní chain, který se skládá ze zvolených kandidátů (*Atropos*), a tyto data nejsou uložený přímo v databázi, ale v databázi konkrétního probíhajícího stavu epochy.

Průběh přenosu probíhá tak, že hostující server si průběžně kontroluje konec epochy a jakmile je epocha ukončena, tak si drží v paměti její snapshot (vždy jen ten poslední). Mezi klientem a serverem se posílají data pomocí webového socketu ve strukturách. Nejprve při navazování spojení je potřeba získat veřejný klíč ze serveru, klient se proto pokusí spojit a zašle vyzývací zprávu (náhodné číslo) ze serveru k němu dostane odpověď - podpis, ze kterého získá veřejný *sepc256k1* klíč. Následně se pouze oznámí požadovaná metoda (stahování dat ze snapshot) a přibližná celková velikost databáze. Následně se začnou rozesílat data z databáze klíč po klíči, po balíčcích, kde je vždy několik klíčů a hodnot, ke kterým jsou vždy dopočítány hashe a podpisy serveru tak, aby byla zajištěna pravost vkládaných záznamů. Tyto balíčky dat jsou zakódovány pomocí RLP knihovny pro převod struktury do bytového pole, následně jsou tyto sekvence zkomprimovány pomocí *lz4* komprese a následně pak jen pro rozlišení jsou komprimovaná data znovu složená do segmentů přes RLP tak, aby klientská strana věděla, kdy má čtená data začít dekomprimovat.

Na klientovi se jednotlivé úseky komprimovaných dat přečtou pomocí RLP, dekomprimují se a uloží se do databáze. Server při takovémto zasílání dat při obdobné konfiguraci není vytížen a největší limit přenosu je zápis do databáze na klientovi, proto server nemá problém udržet krok s nově příchozími block v blockchainu a neopožduje se. Klientovi na konci přenosu stačí jen dohnat všechny nové blocky od momentu vytvoření snapshotu posílaných dat.

[1] `https://github.com/uprendis/go-opera/tree/feature/snapsync-with-ldb-snapshot`

Výsledek je oproti klasickému rsync přibližně 3,5 krát pomalejší při velikosti 800GB, ale jak již bylo zmíněno, tak dostupnost zdrojového serveru není skoro vůbec omezena. Při testování funkčnosti rychlosti pebble databáze, jsem zjistil že je potřeba nahradit nejnovější verzí. Ve staré verzi je totiž chyba, kde při hodně objemném zápisu nebyl správně prováděn `compact` databáze. Pokud nebyl compact volán vůbec trvala synchronizace pro 787 GB přes 36 hodin a 44 minut - 0.358 GB/min. Tuto rychlost jsem vylepšil ručním volání `compactu` každých 5mil. záznamů s databází. Pro databázi o velikosti 832 GB doba trvání synchronizace byla 9 hodin a 21 minut, tudíž 1,48 GB/min, jen se zde čtvrtinu ze zápisu na disk ovšem čekalo na to než se provede `compact`. Rozhodl jsem se proto otestovat novou verzi a zjistil jsem že je již chyba opravená a časově jsem se dostal na 2GB/min. I když největší čekání spočívá v zápisu do databáze rozhodl jsem se vyzkoušet *lz4* kompresi. Ta dobu synchronizace nijak nezpomalila, ale snížila celkový počet přenesených dat o 50 %.

Hlavním důvodem proč je mé kopírování o tolik pomalejší je fakt, že čím větší databáze tím pomalejší zápis dat - při velikosti databáze 85GB byla celková rychlost přenosu 5GB/s, ale při zápisu 800GB byla celková rychlost přenosu 2GB/s. Další proč je přímé kopírování databáze oproti mému řešení rychlejší je ten, že jak pebble, tak leveldb databáze uložené hodnoty komprimují. Databáze dle měření měla velikost jen 62 % skutečných dat. Dalším důvodem je nutná průběžná reorganizace a promazávání již mergovaných záznamů. Výhoda mé implementace spočívá v menším zatížení zdrojového serveru. Rozdíl oproti snapsync synchronizaci je zásadní v tom, že mé řešení data stahuje z jednoho serveru. Co se bezpečnosti hashování a podpisů týče, je třeba ovšem spoléhat na bezpečnost zdrojového serveru. Snapsync s žádnou historií je pro většinu uživatelů nejrychlejší způsob zapojení se do sítě. Pokud ovšem uživatel hodlá transakce na blockchainu vytěžovat a s daty pracovat, například RPC uzel, je pro některé úlohy nezbytné historii buď mít celou a nebo alespoň do nějakého určitého data. Alternativní možností by bylo stažení snapshotu (toto je homonimum - jedná se o vyexportovaný soubor k synchronizaci bez nutnosti použití sítě). Zde ovšem nastává totožný problém vkládání do databáze jako v mém řešení a navíc je potřeba data prvně stahovat a snapshot si buďto od dostatečně důvěryhodného zdroje stahovat a nebo si ho vytvářet. Synchronizace z co nejnovějšího snapshotu je žádoucí, jelikož ze starého snapshotu by znovu mohlo trvat dlouhou dobu se dosynchronizovat k aktuálnímu vrcholku blockchainu.

Na konec bych zde dodal jen to, že moje implementace není nejrychlejší, ale přidává další možnost zabezpečeného a ověřitelného přenosu dat, který by neměl být napadnutelný. Bohužel s rostoucím počtem záznamů do databáze, klesá zápisová rychlost. Všechny přenosu mimo `rsync` jsou tímto zpomalením postihnuty. Co se správosti stažených dat týče, tak aby si uživatel byl opravdu 100 % jistý, že má pravý head blockchainu je potřeba, aby si spustil přepočet všech transakcí od počátku sítě. Což při velikosti sítě 2 TB zabere přes 1 měsíc.

Fast-Sync of New Nodes in the Fantom Opera Platform

Declaration

I hereby declare that this Master's thesis was prepared as an original work by the author under the supervision of Ing. Martin Perešíni. The supplementary information was provided by Ing. Jiří Málek and Ing. Jan Kalina. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

. . . .
Matěj Mlejnek
May 25, 2022

Acknowledgements

I would love to thank my supervisor Ing. Martin Perešíni for his support and consultations. Additionally I would love to thank Ing. Jiří Málek and Ing. Jan Kalina for their help in helping me understand a lot of concepts in Fantom and blockchains in general. Last, but not least, I give my thanks to Ing. Homoliak Ivan Ph.D. for our consultation.

Contents

Chapter 1

Introduction

In light of global inflation, where governments are printing more money and devalue their own currency, the demand for blockchains crypto currencies as the method of payment rises. Blockchains can be used as Distributed ledger system working as alternative to regular fiat currencies. The goal is to create decentralized means of transferring money, where new money can't be printed, so that people can trust that their funds will hold onto their value. Since the first blockchain, the Bitcoin, the number of blockchains and variety of different consenses has increased a lot. Along with a simple token transfers new possibilities of various uses have emerged. Blockchain has audibility, non-immutable truth, verification of, not only financial transfers, but also of certifying the validity of items in our world, NTFs, elections, loans, auctions and so much more.

Alongside Bitcoin, the broadly used, innovative blockchain networks are Ethereum, Solana, Cardano, Avalanche, Fantom, Binance smart chain, Polygon and others. Different technologies have often different positives and negatives, some of which can be improved on, but some, due to the architecture, are hard or impossible to fix. The goal of this Master's Thesis is to improve the *Fantom Opera* blockchain synchronization of new node.

Thesis structure

In the following chapter of this thesis you can read about blockchain technologies and their basic principles such as Proof of Work, Proof of Stake families of consenses, the important concepts and their definitions to help the reader understand what blockchains are. In third chapter I focus on Fantom Opera network and it's specifics. As go-opera implementation is based on go-ethereum (geth) implementation, many things are taken over and are not changed at all, but other, such as the consensus - how nodes decide on new transaction execution (inclusion), is based on Lachesis rather than mining Proof of Work concept in Ethereum or Bitcoin. In the forth chapter, I analyse various methods currently available for synchronization of new node in various blockchains and I introduce my designed solution on how new node in Fantom Opera could be synchronized compared to currently available means of synchronization. In the fifth part, I describe my implementation-specific details, my process of implementation and my thought process and reasons for choosing the way to implement the important segments in code. The sixth chapter is about Experiments and efficiency evaluation of how fast and secure the final designed solution is as well as possible future improvements. The last chapter is my conclusion of how usefull the result is.

Chapter 2

Blockchain

In Public ledger system, where, in general, transactions are put into blocks, which are linked together and creating link. This link of blocks is where the name blockchain comes from. These networks are meant to be public, secure and immutable chains of data that because it is linked together and each information is computed using the data from the previous segment. The main purpose of creating blockchains at first was as usage of decentralized financial systems. In recent years there has been a boom of new means of transferring assets in distributed systems. New implementations of various usages of smart contracts have also helped blockchains (that implement them) to gain on the popularity that it has today. Smart contracts in short are functions computed in virtual machine on each node and give every access to this virtual distributed computer, which can be used to store and compute data securely and immutably.

2.1 Distributed ledger technologies

Most blockchains can be classified as Distributed ledger technologies - DLT, where the system is used as a means of recording transactions of various assets - each blockchain typically has it's own token which is used as a currency. Keeping track of transactions is important so everyone knows who has how many assets. The important properties of these systems are authority, decentralisation, and audibility.

In contrast to the Centralised ledgers in blockchains, there ideally are decentralised consensus nodes, which helps to keep network security and protects against perpetrator's malicious attacks. In centralized systems, the authority is relayed by entrusting some third party to have control over your assets - this is big concern for many people - so blockchains as we know them today were developed. The problem of plain ledger without any security mechanisms is that anyone would be able to modify data. Therefore, cryptographic mechanisms had to be introduced. One of the obstacles for registering transfers is verification if the owner of the asset is the one agreeing with the transfer. Simple signature of order isn't secure because there could be replay (double spending) attacks, therefore mechanism as nonce numbering transaction order was introduced 2.3. Various security improvements led to the creation of Bitcoin blockchains - which focuses on all these problems as well as keeping anonymity (or at least pseudonymity). Read more in section Bitcoin 2.6.

2.2 Blockchain properties

Decentralisation

Decentralization is an important property of blockchains. The decentralized network is ideally 100% available, censorship resistant, immutable and, by design, transparent. Nodes communicating via P2P communication can be spread out all over the world. Unfortunately, because of the fact that many users use the same hosting services (AWS, IBM), the blockchains aren't often as decentralized as they claim to be.

Authority

There can be DLT design to work under someone's authority, but generally the blockchains are demanded to be without authority. When one entity in network holds majority of consensus power and the network is more centralized, the data could be tinkered with.

Audibility and Transparency

Since Blocks are computed by hash of their parent, there was a chain of events that had to have happened. By computing in chain, you can go up to a desired point from genesis or back from the current head to desired block in time. All account states at that time are immutable and anyone can get this history.

Anonymity

Anonymity is in fact only pseodoanonymity. Each individual account has it's own address generated with a private key so their identity is not revealed by any other identifier, but this anonymity level depends on transactions done by that user - for example sending money to or from crypto currency markets, which know identity the user who is transferring the money to be exchanged from a classic bank account.

Permission

- **Permissionless** - the network is publicly available and anyone can join the P2P network, listen to data and emit new transactions.
- **Permissioned** - new node has to be confirmed by authority to join the network.
- **Semi-permissionless** - joining node needs to be confirmed from one of the nodes already in network.

Scalability

For blockchain systems to be usable in practical applications it has to be robust and able to scale processed transactions per second (TPS) as the networks grows. Networks based on *Directed Acyclic Graphs (DAG)* have generally way higher TPS. Another option that is used a lot are multi level blockchains/side chains, where the state of network is validated with higher level only once in a while, therefore lowering the amount of data needing to be confirmed in one blockchain.

Finality and Immutability

Network finality of transactions can vary a lot - varying from a few seconds to minutes. The blocks in blockchains, as they are created from block before, have their finality increased by the increasing depth of block in the chain. The block could be thrown out of network if some other part of the network computes longer chain. The transaction from the original block might not have been included in any of the computed blocks in the other branch. When finality for block is reached, the transactions in it are considered to be irreversible.

Blockchain trilemma

Scalability, security and decentralization are considered to be blockchain trilemma since it gets significantly harder to uphold all three at the same time. Some consenses are better balanced than others, but there is still huge room for improvement as even the most popular networks today often have the transactions limit per second set very low since the network could get congested. Because of this, the price of each transaction often ranges from few cents to tens euros. An attempt to solve one of the three aspects will lead to sacrifice in the other two shown in Fig. 2.1. [8]

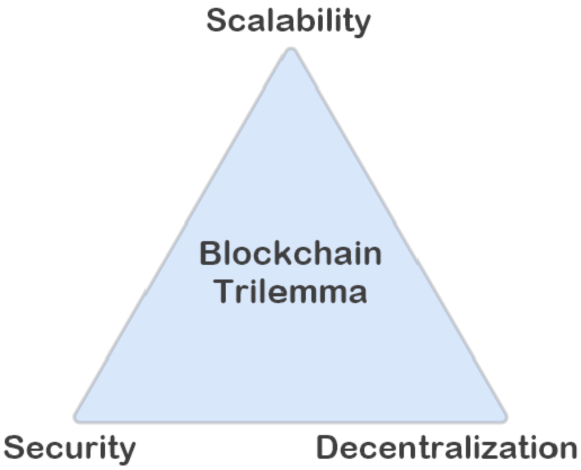

Figure 2.1: Blockchain Trilemma [8]

2.3 Blockchain principles

This section describes terminology and components used in blockchains. Even though blockchain is complex, it can be divided into smaller parts. This part contains useful information for the following chapters.

Stack model

For a breakdown of how blockchains are structured, see *Stack Model* in Figure 2.2. The model consists of 4 main layers: Application Layer, Replicated (Global) State Machine Layer, Consensus Layer, and the last layer is for Data and Network Organization.

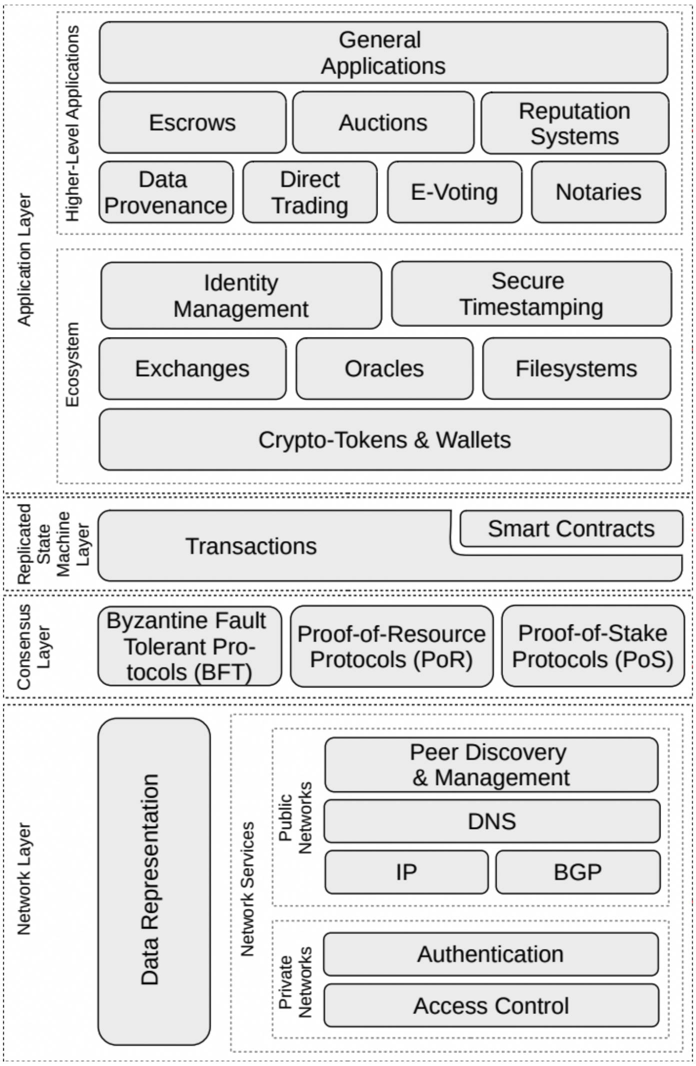

Figure 2.2: Stack Model [5]

Application Layer

The application layer provides interfaces that end users interact with. For example, Distributed Applications, cryptocurrencies, and other functionalities or services. It can be divided into two groups. **Higher-Level Applications** - In this group fall various applications designed for user interaction which are more complex in this category: escrows, auctions, e-voting, direct trading, notaries, reputation systems and general applications for blockchains. **Ecosystem** - this group is for common functionalities such as wallets management, exchanges, secure time-stamping, and oracles. [22, 5]

Replicated State Maschine Layer

In this layer, Distributed Virtual computer is defining how Smart Contracts are Executed in Distributed Virtual Machines. Additionally, there is RSM - replicated state machine that deals with interpretations of transactions. [22, 5]

Consensus Layer

The consensus layer is defining how consensus in given blockchain is reached. For example, it helps with logic of ordering of transactions, rewards for validating blocks, and revealing cheaters. For a more detailed description of how consenses work and what they do see, section 2.4. [5]

Network layer

Network layer is on the lowest level of this model and it consists of network services, storage, encoding, data protection and representation, discovery of new nodes, addressing, routing, DNS and comunication with peer nodes. [5]

Block

A block consists of transaction data, timestamp, signature, nonce - depending on consensus type, the data in them can vary. A new block is linked by containing hash of the previous block that came immediately before it. Depending on the network, one block is usually a few seconds or minutes long.

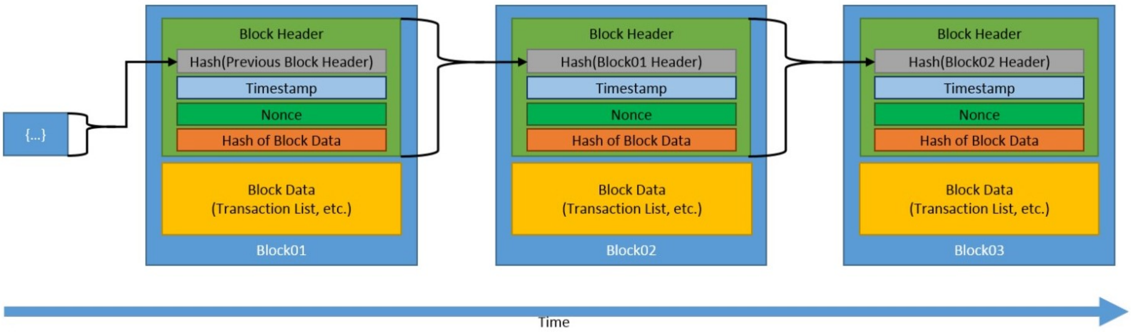

Figure 2.3: Chain of blocks [24]

Epoch

An epoch in general is a time period, which consists of blocks that were issued in it's duration. Epochs can contain data such as list of validators for the next epoch (often in proof of stake consenses).

Transaction fees

In bitcoin, the fees are set as fixed-reward for including your transaction from the transaction pool in computation of the next block, but fees in Ethereum or similar networks are computed based on gas, where the user specifies how much he is willing to pay for each gas unit and what the limit of gas to be used is. For example, simply sending assets from one address to another can cost very little compared to invoking some smart contract function. The estimated gas limit is calculated to the user via RPC and is based on operations predefined costs in Ethereum virtual machine - or alike.

Transaction price is determined by price of gas and it's amount used. If the price for gas is over-paid, then the transaction will be solved sooner, because miners are going to prioritize to compute transactions, which results in higher reward. Overpaying leads to avoidable costs. On the other hand, if you underpay with gas, the transaction will take longer to compute. The amount of gas used differs on parameters such as storage and cost of computation. Different functions have a different price. For example, an easy lookup function can have a way higher price if data is cached or not. For estimating gas price and units of gas *Oracle* is used. For example, a transaction can have a limit of 21 000 units of gas as well as specification of how much the user is willing to pay for each each unit of gas. [23]

EVM

Ethereum Virtual Machine - EVM created for Ethereum enables executable transactions in blockchains. EVM instruction set for code execution in distributed systems, this provides decentralized processing of transactions with transparent processes. This distributed system is secure, because data is stored directly in blockchain. EVM is stack based with random access memory and persistent storage. Instructions in EVM have assigned number of gas required for the specific operation so when code is run user has to specify how much gas is he willing to use. This gas mechanism is used to pay processing time as well as to avoid infinite loops by kicking out programs, which use up all their gas. Because it is decentralized it doesn't have disadvantages of centralized systems, where is a single point of failure, censorship of clients and less them 100% availability.[4]

Smart contracts

On top of basic interface of just transferring assets, the network can implement an extended set of functions. For example, when used on voting or any other specific usecase for a given network - these networks are generally built on consenses and are faster - for example, creditcard payments can be done with preset list of functions. The biggest drawback is that when a network wants to support new functionality, most nodes in the network (depending on consensus) have to be running an updated version of software. This process is generally very slow and it's hard to keep track of everything because there might be some party requesting functionality, but majority of network might consider this functionality not as relevant and might not be incentivised to add it to it's codebase. Because of this fact, the drive of generally any code running code in virtual machine got into the foreground. Smart contracts are originating in *Ethereum EVM*, where each node is computing the state of variables in it's local storage (except for specialized low requirement nodes). These smart contracts have interface so others can access it. Smart contract has various uses, such as implementation of fungible and non-fungible tokens, auctions, wallets, loans and many more. As an example, I would like to mention multi-signature contracts they are used to hold funds and interact with other contracts, creating a middle man between the set of users and the interacting contract, which, depending on set policies, demand at least some number of owners approving on that transaction. This prevents a smaller portion of shady users from withdrawing all the funds from shared accounts into their private accounts.

The two most used programming languages for smart contracts are *Solidity* and *Vyper*. Solidity is a high-level, object-oriented, statically typed programming language with very easy to orient C++ / JavaScript like syntax. Vyper is python-alike language, which doesn't have as many features as Solidity but it's main purpose is to be easier to audit. As a new contract is deployed onto the network, it has it's specific address towards which implemented methods can be called. Each function can have modifiers; these modifiers can be used to establish if the caller of function is eligible for calling this method and meets all necessary requirements. For example, only the deployer of the contract can be set to mint new tokens in NFT contract and so on. Methods are called via ABI in which conversion of the function identifier and parameters are converted to binary code. [7, 1]

Smart contract address

The deployed contract on the network gets it's own specific address computed from the owner's public key and his nonce to it. When the contract is being deployed on multiple blockchains at once, and the owner keeps the same `nonces`, then the contracts will have same address.

ERC standards

`ERC` - **Ethereum Request for Comments** is an extension of an `EIP` - **Ethereum improvement proposals** - defining standards on Ethereum platform. ERCs are application-level standards and conventions are used to define bounds on proper usage of smart contracts on how they have to behave. They are used, for example, for token standards *(ERC-20, ERC-223, ERC-721, ERC-777)*, name registries *(ERC-137)*, URI schemes *(ERC-681)*, library/package formats *EIP190* and wallet formats *(EIP-85)*. When it comes to tokens, they can be distinguished into two main categories depending on what they represent - either Fungible (*FT*) or Non-Fungible tokens (*NFT*). Fungible tokens are representing something which is of some type indistinguishable from others such as currencies. On the other hand, Non-fungible tokens represent something unique. The following standards describe functionality of token handling and their transferring API interfaces. [17] Creation of new tokens is called minting - the data, for example, image hash, has it's representation registered into the blockchain. NFT - non-funguble token representation - the data of the item (image) itself is not uploaded to the blockchain, but there is some link there such as a link for *IPFS - InterPlanetary File System.*

- **ERC-20** - Contracts deployed with this standard represent one type of token (for each new type there has to be a new contract).
- **ERC-721** - Nowadays, one of the most frequently used non-fungible token standards. One contract consists of a collection of distinct token IDs.
- **ERC-1155** - An improvement of ERC-721 tokens as it can be used to create fungible and non-fungible token types. These types can be added to an existing smart contract so there is no need for creating a new smart contract. Each token ID has link to metadata which specifies token type.
- **ERC-2981** - Was created as guidance for royalties solution for ERC-721 and ERC-1155. The main objective is to enforce that the trading contract obey royalties set by tokens.

MPT - Merkle Particia Tree

MPT is used for membership proofs see Figure 2.4. Every single node in tree data structure contains hash of nodes bellow. It can be used to efficiently verify that some specific transaction was in a block, not by recomputing the whole tree, but just the path to the main root. Leaf node contains data. None leaf node contains hash of children.[25]

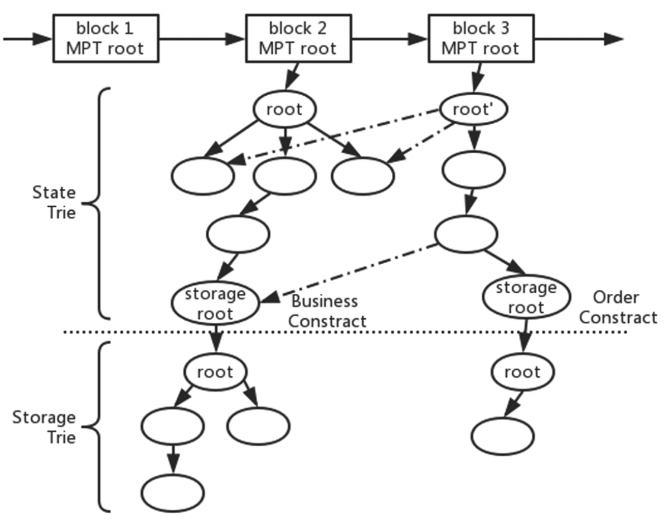

Figure 2.4: MPT tree in Ethereum. [25]

Nonce

There are (at least) two different usages of nonce. One type is one of the block's attribute values, where nonce here can be used as complementary value for a block to have specific hash, for example, in bitcoin. The second usage is to effectively counter each transaction for a specific address, then nonce is implemented via the Lamport rule. This is useful as prevention of the replay attacks since each transaction of the account is bound to have different hash because the nonce has increased after the transaction was completed.

Interfaces

Through the following interfacing you can send data trough blockchains, but also listen to and query for data in it. Methods in Smart contracts are called ABI - Application binary interface, which is essentially just a stream of bytes converted from method name and calling arguments.

- **RPC** - Remote Procedure Call is interface that is used to communicate with the blockchain network from outside of the node. It is often operated on a regular node in the network. You can, for example, use graphql in standard http request or by web3 opening websocket, which can be additionally used for data subscriptions.

DAG

DAG - Directed Acyclic Graph is another option on how blockchains can be linked together. Transactions are not grouped one by one and processed as blocks, but are computed individually. There is often some sort of mainchain towards which other transactions confirm to - hence better scalability can be achieved. Blocks in DAG typed networks can still exist, but they are used as headers for setting time boundaries in blockchain and carry additional confirmations(list of validators, reconfirmations, gas used). These types of network generally have lower fees associated with costs of transactions because they typically use either small scaled PoW for each event (IOTA), or PoS(Lachesis, Avalanche) for reaching consensus.

Lamport rule

The Lamport rule is an algorithmic logical clock which determines the order of events in distributed systems. This algorithm is based on counters, where each node has its own counter (usually multiple counters for different types of messages). Each process has counters which increment with every new created event by one and with newly received events from other processes - so the order of each event can be distinguished due to causality. We have to consider what it takes for the clock in each system to be correct. We can't base our clock on physical time, because that would require keeping a timestamp of every event and our decision on when events occurred has to be based on actual order. The most important condition is that if one event occurred before another event, then it has to be computed before that event. [10]

Attacks

Double spending

The double spending attack is linked to a finality in blockchain. As finality is reached, the chance for double spending is negated. It works by attempting to creating forks in the chain so that the funds can be seen as transferred, thus resulting in contract execution. Then this action gets reverted because the other fork won - the longer chain was computed quicker. For example, when bridging assets from one blockchain to another, there have to be bigger delays and additional waiting time, making sure that finality is truly reached.

51% Attack

51% Attack often requires huge computational power to overwrite confirmed transactions. Since transactions get confirmed based on finality of the network as no block is confirmed on 100%, then its finality gets exponentially harder to be overwritten.

DoS Attack

Denial of service attacks have different forms. A successful DoS attack on a consensus node will be resulting in lower network consensus power, therefore preventing nodes from being rewarded. In PoS systems *Validator* nodes never intentionally reveal whether you are communicating directly with them. Their private key for signing transactions is, of course, different to the P2P key that is used for sending events to peers. By network analysis in smaller scaled networks, it would be possible to find out if the node you are communicating with is a validator. If more than 1/3 of all validator's power is lost in BFT network, then it would be at a halt. [5]

2.4 Consensus mechanisms

In simple terms, consensuses can be seen as the agreement protocol between nodes in a P2P network. Distributing and processing network data in a way that the consensus is reached when consensus nodes agree on the correct state of the network. The first of two main consensus families of consensuses mechanisms are Proof of Work - PoW, where consensus is reached thanks to consumption of some resource mostly using computational power 2.4. The second type of reaching consensus is Proof of Stake - PoS, which is based on nodes having different consensus power depending on their staked asset value. See 2.4.

Consenses in blockchain are used to achieve agreement between nodes. Consensus is a way nodes come to agreement about new transactions. There are various types of consenses based on various things like computational power, staking value for validating, proof of history, proof of place and many others. Generally, whenever a node helps the network reach consensus about something, then there is some reward associated with the help. This helps the decentralisation of the network.

Nodes which help a network reach consensus are called consensus nodes, then, in Proof of Work type of networks, they are called miners and, in Proof of State networks, they are called validators. As they help with new blocks being emitted and confirmed, they are getting rewards for the new block not only from the network itself, but also often from fees payed by users. The higher-paid the transaction, the faster it is going to to be picked up from the transaction pool and it will finish faster.

Proof of Work

Proof of work, as mentioned in the previous section, is based on computational power. There are also other variants in this family using other types of resources, for example, proof of storage.

In Bitcoin, there is Nakamoto's consensus based on hard computational problem. Every new block has nonce added to it so that the resulting hash will lead with zero bits. The amount of work needed to be done to find hash with a bigger amount of leading zeroes is exponential. Each node has its local chains with confirmed prefixes, but the newest blocks don't have to be confirmed yet - delayed finality. There can be two blocks emitted at same time and then there is race in which one that is ahead will be computed further. The other fork is then discarded. PoW, by its design, has to be hard to compute or expensive on resources so it is hard for malicious actors to join in on computation and try to tinker with the network's finality - because of this there is the reason why Bitcoin and other PoW networks often wait a bit longer for finality to be confirmed. To regulate the rate of new blocks being generated, the difficulty of the computational hashing is updated every 2016 blocks.[2, 11] The Bitcoin network, with its miners, consume in electricity consumption together more than many smaller counties. This ecological fact for these PoW networks is a reason that they are slowly in decline as more and more people are pushing for more and more sustainable means of power consumption. The electricity used for bitcoin mining rigs is often not from renewable sources, but from sources that are harmful for the environment. The amount of CO2 emissions by bitcoin in year 2021 is estimated to have killed about 19,000 people. [20]

Proof of Stake

Proof of Stake is consensus family, where consensus nodes are called validators who have to put in their own funds as collateral and then, depending on total percentage of their share, they get according amount of voting power and can vote on newly emitted events. Voting is basically computation of transaction result and verifying that the transaction can be completed without violating network state. If a validator behaves maliciously, their funds can be slashed to zero and they can be removed from the list of validating nodes. As new blocks are emitted, validators that contributed with their signatures are rewarded with fees.

Stacking

Stacking is when client not running his own node entrusts some validator with his assets in the PoS network. This mechanism of stacking, where people who don't want to run their own validating nodes help to contribute to network security by letting people choose whether to and where to delegate their funds. Then the selected validator gets to increase their validating power by stacking an amount of clients.

Nakamoto consensus

This consensus is reached based on computed hash function from newly included transactions in new block. The computation is based on finding number which in addition to the block information will result in hash in some specific form. When consensus block (miner) computes hash for new block he sends out his findings to his peers and is in result rewarded for contributing to emitting this block. This method is very slow and as there is limit of how many transactions can be computed at once so price for transaction included in new block can get high. Miners are rewarded with this fees as well as rewards of emitting new block implemented in Bitcoin code this reward is halved every 210000 blocks by reducing this rewards the total number of bitcoin is increasing smaller and smaller amount, by the year 2024 the emitting of new blocks will depend on transaction fees alone as the smallest bitcoin unit will be reached and no more bitcoins won't be mined into circulation [21].

Lachesis aBFT

Lachesis aBFT is P2P consensus used in Fantom Opera. In contrast to other consenses Lachesis distributes events between users rather than states themselves. Each transaction can be composed of different number of events. These events are stored in DAG - direct acyclic graph. Input of each node are Events, which as they move trough network are signed off by the validators. The signing mechanism is based on `Proof of stake`. Once more then 2/3 of network total stake agrees on event (signs it) it is considered correct and consensus about that event is reached. Therefore there isn't block as we know it from PoW (being confirmed as a whole object together), but the transactions themselves are only afterwards bundled up into blocks for checksum of network state. In the implementation of the consensus itself the events are being continuously ordered and output is the final order of events (the output of consensus is also list of cheating validators. Ordering is deterministic implemented by lamport rule 2.3 - so each node comes up with same solution. At the end of each block epochs can be also sealed. Epochs in *Lachesis* are important bounds for Atropos calculations. There are two implementations of Lachesis - one of which is `go-lachesis`[1] written in Golang, this version is also used in `go-opera`. The other is `jlachesis` written in java [2], which seems to be abandoned. There was an experiment with creating network on Lachesis protocol with hardcoded set of transactions reaching 30,000 transactions per second.

[1] `https://github.com/Fantom-foundation/go-lachesis`

[2] `https://github.com/Fantom-foundation/jlachesis`

2.5 Node types

Depending on the specific blockchain options and user preference he can choose between different types of which node he wants to run. Sorting is either based on the type of job it has - validator/miner node or just ordinary node for getting statistics from history and listening to current events happening on blockchain and scanning for information. Another sorting is by the amount of information the node stores and listens to. Figure 2.5 represents how different types of nodes interact with blockchain.

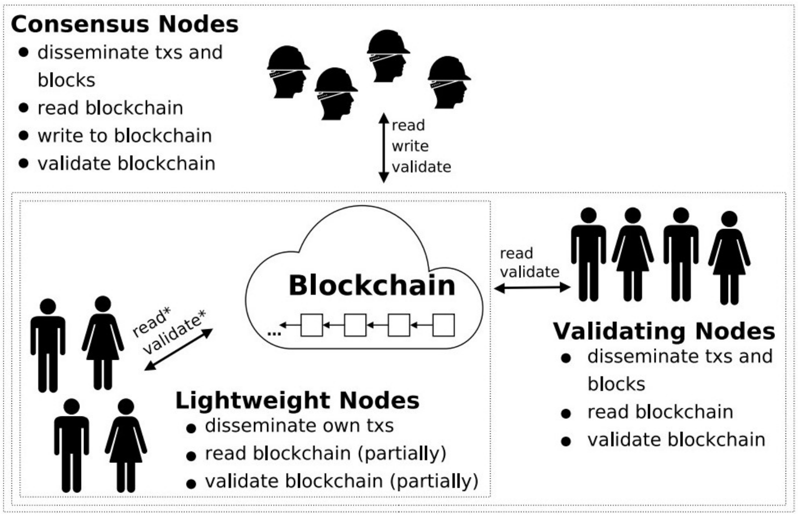

Figure 2.5: Involved parties with their interactions and hierarchy. [5]

They can be sorted by amount of information they are holding. Since not every node needs to have full history of every transaction. Most of the nodes just want fast access to be able to send transactions to the blockchain directly.

- **Lightweight node** - In contrast with regular ethereum nodes they store just head of the blockchain. Usually used on devices with low computing power.
- **Archive node** - storing all data and states.
- **Full node** - storing whole state MPT in Ethereum
- **Pruned node** - similarly is light node storing just recent data up to some point.

2.6 Blockchain networks

Bitcoin

Bitcoin is decentralized P2P network, that was first proposed in year 2008 by Satoshi Nakamoto. This blockchain system adapts cyptographic mechanism, that enable anonymous peers to complete their transactions. It uses *Nakamoto consensus* described in section 2.4. Bitcoin miners get together in pool, which is coordinating their computational power. As mining pools introduced and block mining difficulty keeps getting harder - it is practically impossible to mine block with smaller computational power - first confirmed block taking years to occur. [21]

Ethereum

Ethereum is distributed blockchain allowing smart contract execution. Smart contracts is program, which when deployed is running and computing it's code on nodes in network in *EVM - Ethereum Virtual Machine.* Two mostly most widely used implementations connecting to Ethereum protocol are *Geth - go-ethereum* written in Golang and *Parity - parity-ethereum* written in Rust [3]. The concensus in Ethereum is based on Proof of Work. [1] Ethereum is currently second network in total capital value and because of the many innovative features it is used as template for many other networks. Ethereum is decentralized and anyone can join the network and become miner. In order to mine new block node has to find nonce that corresponds with transaction data into specific hash. [15]

Ethereum 2

Aims to be more scalable then Ethereum and public testnet is already running. Uses sharding of network to help with scaling. Synchronization between shards is done trough beacon. Currently troughput of network is . but we will have to see how this will work in mainnet with way more shards. As long as transaction is just in between bounds of shard it is very quick, but beacon's bandwidth is going to be bottleneck for a lot of transactions (which need to access data from other shards).

Cardano

Cardano is as well Ethereum blockchain with Smart Contracts ability. It ss project which aims to formally verify implementation of `Ourboros` family consensus protocols. Cardano's consensus*Outrobos* uses Proof of Stake mechanisms is focusing on scaling transactions. [6]

Layer 2 blockchains and sidechains

In layer 1 blockchains such as Ethereum, Cardano, Fantom all transactions are computed on network itself, which is limiting because there is limited amount of transactions that can be computed per second. Layer 2 blockchains offer the ability to compute the transactions and then periodically confirm their state with first layer, thanks to this mechanism they can be more scalable and therefore can also have lower the transaction fees. Concurrently with blockchain processing transactions offloading transactions from mainchain, get sum of results and then periodicaly process/verify it on mainchannel. For example Hydra for cardano, Plasma for Ethereum

[3] `https://github.com/openethereum/parity-ethereum`

Chapter 3

Fantom Opera

This chapter describes what Fantom Opera is. Fantom is one of the first widely used DAGs. In this network transactions are sent in form of events by *Lachesis* protocol, which ensures that they are valid. Lachesis provides fast, leaderless, asynchronous, permissionless , quick finality, cheap transactions and also fast. Various decentralized application be built onto it, but also since Opera uses Ethereum Virtual Machine for smart contracts, the code from Ethereum can be portable without any problems. [3]

Opera

Opera is Fantom's mainnet network name it is running since december 2019. The first block in blockchain is called genesis, from this block onward events, transactions, contracts, validators constantly change the state of header block of the Fantom network. Currently there is only one implementation for Opera[1] in Golang, where Opera is built of top of Ethereum Geth but with with Lachesis consensus2.4. For datastorage it uses leveldb database for fast key-value writes and searches.

Networking

When new node is synchronizing to the network it loads list of bootnodes in configuration file, which client asks for IP addresses to connect to. So they don't necessarily have to be running nodes on that address. Then when node starts communicating with another nodes they keep peer connection between each other. In future when node needs to connect it looks in history of connected nodes(peers) and tries to connect to them.

Event blocks are being sent trough gossiping emission speed is based on amount of traffic in the network.

Blocks

Blocks in lachesis are not just bundles of transactions. One block also consists of all of the intertwined events in DAG and since in next block events from previous blocks are needed as well. So you can't just take one block and start computing viz. fast sync.

[1] `https://github.com/Fantom-foundation/go-opera`

```
type Block struct {
       Time Timestamp
       Atropos hash.Event
       Events hash.Events
       Txs []common.Hash
       InternalTxs []common.Hash
       SkippedTxs []uint32
       GasUsed uint64
       Root hash.Hash
```

Listing 3.1: Block structure

Transaction pool

Transaction pool - Tx Pool waiting queue base on every node for new transactions emission. Based on the amount of gas price, the priority of transactions is defined and when validating node takes transactions from the pool in batch new block is created. [14]

Epochs

Difference with general purpose of epochs described in 2.3. DAG structure of events represents single event structure. DAG is separated into sub-DAGs - which are considered `epochs`. Epoch creates is bounds in for blocks in time. Epoch is extended block, that has additional information such as list of validators.

Epoch ends after on of three conditions is satisfied:

- number blocks exceeds `MaxEpochBlocks`
- time of `MaxEpochDuration` is reached
- block in which cheater was revealed ends

DAG

DAG model is integrated un Lachesis protocol. It enables high speed stream of asynchronous data. The data stored in DAG are events, where Lachesis concensus helps with their ordering, this asynchronous ordering with logical time ordering is used instead of classical chain in blockchains. DAG in Opera is *StakeDag* aiming for consensus in Proof of Stake network by DAG-based trustless system. Validators have higher score of trust and usual users have low score of trust. The Asynchronous approach of BFT system boardcasts events to be voted on. Each validator votes for these transactions to be signed as 2/3 of network agrees concensus is reached. [14, 13]

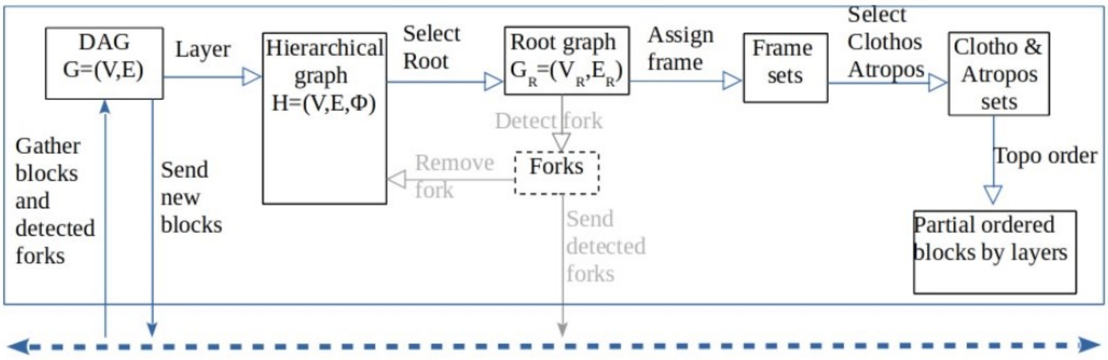

Figure 3.1: An overview of ONLAY framework[12]

- **Layering** Layering assigns every block in OPERA chain a number with usage of Lamport rule so that every edge only points from old to new layers.
- **Root** - Root is an event block is called a root if it is the first generated event block of a node, or it can reach by more than 2/3 of other roots.
- **Root** graph - Root graph is containing roots as vertices and their reachability between roots as edges.
- **Clotho** - A Clotho is a root node in DAG structure satisfying that it is known by more than 2/3 nodes and more than 2/3 nodes know that information.
- **Atropos** - Atroposes are used to keep main line of DAG they are selected Clotho by voting and they have consensus time. This helps with ordering (=finality) of the blocks in network. Starting new epoch there is no reference to the events in dag before, but as the epoch carries on including new blocks there are backward references in DAG, which helps to scale up the network. [12]

Snapshots

Nodes which have enabled this feature are creating it's own snapshot of data, which is automatically updated with new data. This storage consists of keys in flat storage so whenever contract specific address is being searched the lookup time is significantly smaller.

This feature was created in Ethereum - last year there was possible exploit - when smart contract code was accessing the non cached data, then it took few seconds to access it and there wasn't any bigger gas penalty. A bigger penalty for accessing non cached data was added. So still when data to a key is being searched and there is flat storage it returns data for lower amount of gas.

Storage

Data for Opera stored in levelDB. The structure of files in filesystem:

- **/chaindata/genesis** - database used when loading data from genesis file
- **/chaindata/gossip** - database with events

- **/chaindata/gossip-X** - data for specific epochState
- **/chaindata/lachesis** - concensus database
- **/chaindata/lachesis-X** - data for specific epochState
- **go-opera** - private key for P2P communications and list of bootnodes
- **keystore** - private key for used for signing transactions
- **opera.ipc** - inter-process communication

Gossip database structure

Data stored in database is compressed and stored in file system in structure shown in previous section. Data is stored by prefixes of legth of one or two characters decoded by hex so 16 bits. In Table 3.1 you can see amounts of data and data their data distribution in **mainDB gossip** database. Database size was **1520.3GB** (1520378895738 Byte bytes). The fact various types of data are going to have different sizes of keys and values, therefore database usage can be more overloaded as multiple amount of small writes or reads in database can lead to decrease in total write/read limits.

Prefix	Node	count (Byte)
_	gossip.Store.Version	1
Ds	gossip.Store.BlockEpochState	24057
e	gossip.Store.Events	322195847864
b	gossip.Store.Blocks	8423290404
g	genesis	44
gi	gossip.Store.Genesis.GenesisBlockIndex	1
gg	gossip.Store.Genesis.GenesisHash	0
lk	gossip.Store.HighestLamport	1
V	gossip.Store.NetworkVersion	10
B	gossip.Store.BlockHashes	1597837650
!	gossip.Store.LlrState	20
@	gossip.Store.LlrBlockResults	53098116
#	gossip.Store.LlrEpochResults	243201
$	gossip.Store.LlrBlockVotes	148011005
%	gossip.Store.LlrBlockVotesIndex	75114408
∧	gossip.Store.LlrEpochVotes	8944642
&	gossip.Store.LlrEpochVoteIndex	328650
∗	gossip.Store.LlrLastBlockVotes	469
(	gossip.Store.LlrLastEpochVote	561
S	gossip.Store.SfcAPI	3406873
M	evmstore.Store.Evm	471335184406
r	evmstore.Store.Receipts	163024427797
x	evmstore.Store.TxPositions	20584921251
X	evmstore.Store.Txs	3489287017
L	evmstore.Store.EvmLogs	529030636788
l	HighestLamport	6
h	BlockEpochStateHistory	408058966
P	EpochBlocks	231530

Table 3.1: Each prefix with different amount of data

3.1 Code structure

Structure of Opera processes when processing new events from network is visualized in Figure 3.2.

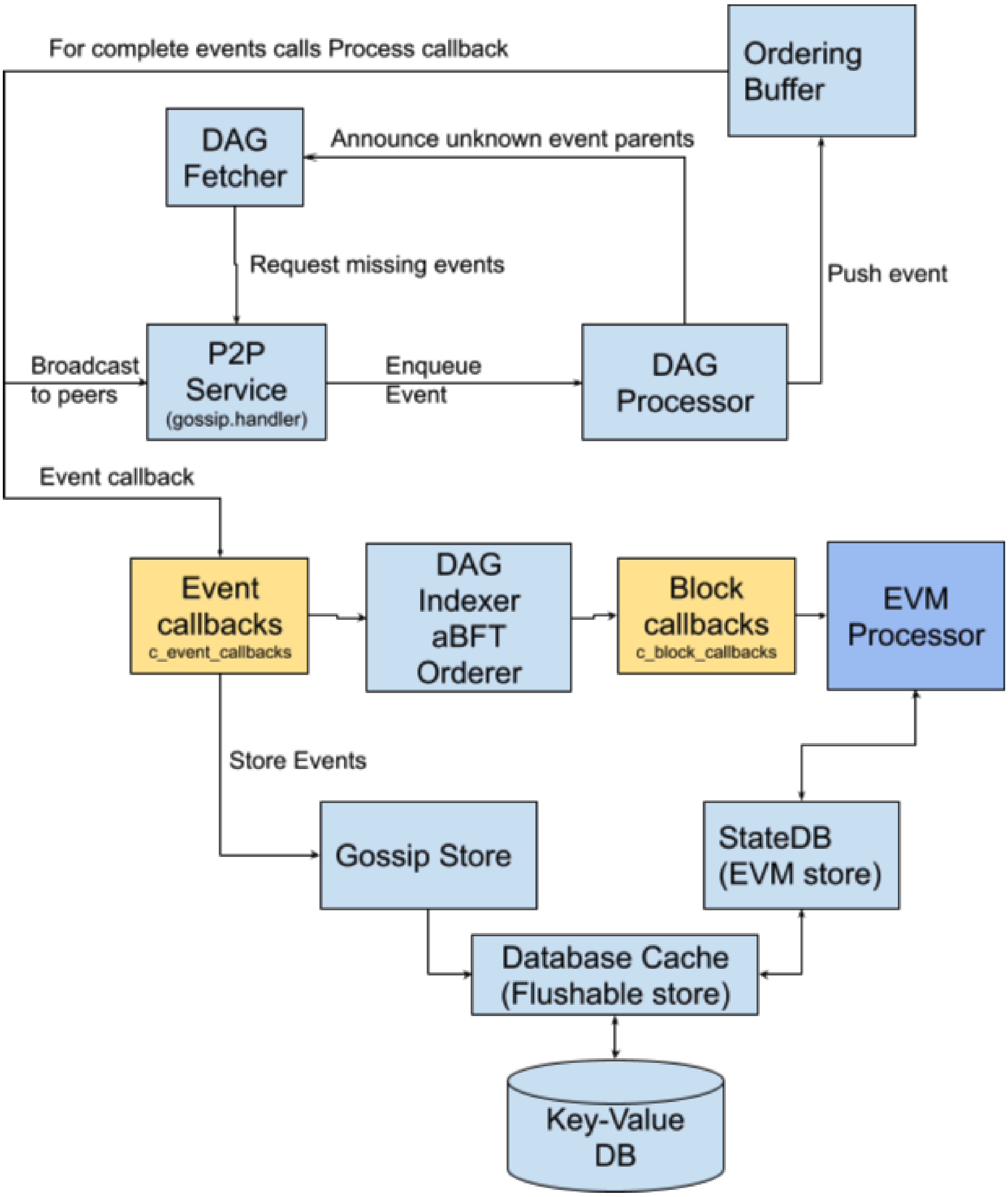

Figure 3.2: Opera structure

- P2P Service (implemented in gossip.handler) handles peers connections and received messages
- Received events are queued to DAG processor's checker and orderedInserter workers
- checker validates events in LightCheck and in parallel in HeavyCheck (heavy_check.go)
- orderedInserter receives validated events, release invalid and push valid events into the Ordering Buffer
- The Ordering Buffer returns a list of parent events, which are required yet - the DAG processor announce them to the DAG Fetcher

- The DAG Fetcher requests the missing events from P2P peers
- The Ordering Buffers calls Process callback on completed events, which (in gossip.handler) broadcast them to peers and calls Event callback (in c_event_callbacks) processEvent.
- Event callback checks the epoch, process LLR votes, saves the event and sends it into the DAG indexer/aBFT Orderer.
- aBFT Events Orderer finds the atropos of added events and construct an ethereum-like block by calling Block callbacks BeginBlock/ApplyEvent/EndBlock.
- The EndBlock callback starts the block processing in EVM processor. It seals the epoch before, if appropriate, and generates internal transactions which awards the epoch validators.
- The transactions processing is done in a standalone blockProcTasks worker thread.

Lachesis base

Lachesis package

- The Lachesis package defines an interface of the consensus implementation[2]
- Input of consensus are events (dag.Event) received from various nodes. Each node receives and emits events. Validators are used to sign events
- Output of consensus are blocks of events in final order and list of cheating validators
- lachesis.Consensus - input interface for consensus implementations
- lachesis.Consensus - input interface for consensus implementations
- Build(dag.MutableEvent) is used by validators to set consensus fields (frame) of new events
- Process(dag.Event) is used to include received events into processing - parents needs to precede their children
- lachesis.ConsensusCallbacks - interface for output callback (of blocks) from the consensus
- BeginBlock / ApplyEvent / EndBlock - used by consensus implementations to output constructed blocks and their events
- lachesis.Cheaters - output type - cheating validators list

[2] `https://github.com/Fantom-foundation/go-lachesis`

aBFT package

- aBFT = Asynchronous Byzantine Fault Tolerance contains implementations of interfaces from the Lachesis package
- abft.Orderer - Processes events to reach finality on their order, does not detect forks/cheaters.
- Calls ApplyAtropos callback to report new decided frame (frameId+atropos hash)
- Calls EpochDBLoaded callback to report start of an epoch (on an epoch sealing or on start)
- abft.Lachesis - wraps Orderer, adds:
  1. Setting events as confirmed
  2. Forks/cheaters detection (needs to read dagIndex for it)
  3. Calls Consensus-Callbacks to output constructed blocks
  4. abft.IndexedLachesis - wraps Lachesis, writes built/processed events into dagIndex, currently used in tests only
- lachesis.ConsensusCallbacks - interface for output callback (of blocks) from the consensus
- BeginBlock / ApplyEvent / EndBlock - used by consensus implementations to output constructed blocks and their events
- lachesis.Cheaters - output type - cheating validators list

dag package

- dag = Directed Acyclic Graph defines interfaces for Events - basic building unit of the DAG
- dag.Event
- ID consists of: = epoch + lamport + counter on validator
  1. epoch
  2. lamport
  3. counter on validator
- Lamport = maximum of parents lamports + 1
- Creator = the validator emitting the event
- Parents = list of parent events
- SelfParent = parent event from the same validator
- Seq = self-parent seq + 1 (0 if it is the first event of the validator)
- Frame = set by abf.Orderer.Build()
- Epoch

VecMT package (vector clock with median time calculation)

VecFC package (vector forkless cause)

Kvdb package (key-value database)

- kvdb.Store = interface for writable key-value store
- Interfaces:

  1. Has/Get - reading
  2. Put/Delete - writing
  3. NewBatch - batch writing (primitive transaction)
  4. NewIterator - iterate data by key prefix
  5. Stat - get stat property (impl specific, like "leveldb.alivesnaps" for number of alive snapshosts)
  6. Compact - flatten the store in key-range (deleted/overridden data are discarded)
  7. Close

- **leveldb.Database** - kvdb.Store implementation using leveldb from go-ethereum
- table.Table - wraps kvdb.Store, adds key-prefix to create nested Store - implements kvdb.Store
- NewTable(prefix) creates table nested in the table (prefixes are appended)
- table.MigrateTables - method for "table" structs initialization - used in data-specific stores bellow
- Initialize structure fields to table.Table instances.
- Argument 1: pointer to structure to be initialized (with "table:" annotation defining the prefix)
- Argument 2: the underlying kvdb.Store
- abft.Store - store for aBFT (not an implementation of kvdb.Store - data-specific methods instead)
- table/epochTable sub-structs are initialized using table.MigrateTables() to sub-tables of mainDB
- epochTable.Roots - Roots specific store methods:

  1. Key - concatenation of frameId + validatorId + eventId, no value
  2. AddRoot(selfParentFrameId, event) - puts into the table
  3. GetFrameRoots(frameId) - iterates with prefix filtering the frameId

- epochTable.ConfimedEvent - maps event hash to id of frame on which was the event confirmed
- epochTable.VectorIndex - subtable for vecengine.Engine subtables (dagIndex data)

- table.EpochState - one constant key “e”, the epoch state (epoch id + validators) in RLP
- table.LastDecidedState - one constant key “d”, RLP encoded value
- vecengine.Engine
    1. table.EventBranch - maps event id to branch id (1 validator has 1..N branches)
    2. table.BranchesInfo - constant “c” to BranchesInfo struct in RLP:
        - BranchIDLastSeq - maps branch id to highest event Seq in the branch
        - BranchIDCreatorIdxs - maps branch id to validator id
        - BranchIDByCreators - validator id to list of branch ids
    3. vecfc.Index - dagIndex - engine/index to detect forkless-cause condition
    4. table.HighestBeforeSeq - maps event id to vector of lowest events (seq) which observes the event
    5. table.LowestAfterSeq - maps event id to to vector of highest events (seq+isForkDetected) which observes the event

Snapshot

- Opera stores state of accounts (balances, contracts data) in MPT (Merkle-Patricia tree), which is necessary to generate hash of the block state. The snapshots can be enabled as flat-storage cache to accelerate access to state data. When snapshots are enabled, a new snapshot is created for every new block (in StateDB.Commit)
- There are two types of snapshots:
    1. diskLayer - the base
    2. diffLayer - collection of modifications made to a state on the top of the other snapshot
- The Cap operation flattens all layers under a given level into the bottom one

Key-value database

- All Opera data is stored into a few key-value databases. If it is not set to use “in-memory” database, it use leveldb to store data in “chaindata” subdirectories
- Connections into both are provided by [integration.DBProducer]
- This producer is wrapped by [flushable.SyncedPool], which ensures:
    1. Writing dirty flag (into FlushIDKey) before flushing and its clearing after the flush
    2. Keeping the databases opened in the pool (opened on first OpenDB call)
    3. When the engine starts [integration.MakeEngine], it checks if some database is dirty (flushID is not written) - if it is, all DBs should be dropped
    4. The flush is done: (for gdb=gossip db) [gossip.Store.Commit]
    5. On new epoch [c_event_callbacks.processEvent]

6. On gossip.Service stop
7. By ticker when cfg.MaxNonFlushedPeriod is exceeded [gossip.Store.Init]
8. After genesis application [integration.applyGenesis]

LevelDB options

- Following LevelDB options can be significant for the database performance:
- CompactionTableSize - size of new table files
- CompactionTotalSize - maximum total size of all tables in one layer, when exceeded, the compact operation starts to merge the layer into the higher layer

KVDB prefixes structure

- Following databases can be found in the “chaindata” directory.

Tag	Namespace	Note
_	gossip.Store.Version	(“id” - string)
D	gossip.Store.BlockEpochState	(“s” - BlockState, EpochState)
e	gossip.Store.Events	(eventID - inter.EventPayload)
b	gossip.Store.Blocks	(blockID - inter.Block)
g	gossip.Store.Genesis	(“i” - blockID)
l	gossip.Store.HighestLamport	(“k” - lamportUint32)
V	gossip.Store.NetworkVersion	(“v” - networkV, “m” =- missedV)
B	gossip.Store.BlockHashes	(eventID - blockID)
S	gossip.Store.SfcAPI	
SR	sfcapi.Store.GasPowerRefund	
S1	sfcapi.Store.Validators	(epoch+stakerID - sfcapi.SfcStaker)
S2	sfcapi.Store.Stakers	(stakerID - sfcapi.SfcStaker)
S3	sfcapi.Store.Delegations	(deleg+staker - sfcapi.SfcDelegation)
S6	sfcapi.Store.DelegationOldRewards	(delegator+stakerID - amountInt)
S7	sfcapi.Store.StakerOldRewards	
S8	sfcapi.Store.StakerDelegationsOldRewards	(stakerID - amountInt)
r	evmstore.Store.Receipts	(blockID - []ReceiptForStorage)
x	evmstore.Store.TxPositions	(txHash - blockID,eventID)
X	evmstore.Store.Txs	(txHash - types.Transaction)
L	evmstore.Store.EvmLogs	
Lt	topicsdb.Index.Topic	
Lr	topicsdb.Index.Logrec	
M	evmstore.Store.Evm	(EvmKvdbTable)

Table 3.2: Caption gossip mainDB (OperaStore) (gdb)

Tag	Namespace	Note
c	abft.Store.LastDecidedState	
e	abft.Store.EpochState	

Table 3.3: Caption mainDB lachesis (LachesisStore/cdb=consensus db)

Tag	Namespace	Note
Z	gossip.asyncStore.Peers	

Table 3.4: Caption asyncDB gossip-async (gdb)

Tag	Namespace	Note
c	genesisstore.Store.Rules	(“c” => opera.Rules)
b	genesisstore.Store.Blocks	(blockId => genesis.Block)
a	genesisstore.Store.EvmAccounts	(address => Balance,Code,Nonce,SelfDestruct)
s	genesisstore.Store.EvmStorage	(address+keyHash => valueHash)
M	genesisstore.Store.RawEvmItems	(keyBytes => valueBytes)
d	genesisstore.Store.Delegations	(address+validatorId => genesis.Delegation)
m	genesisstore.Store.Metadata	(“m” => genesisstore.Metadata)

Table 3.5: Caption genesisDB genesis

Tag	Namespace	Note
t	gossip.epochStore.LastEvents	(“” => (validatorId+eventHash)*)
H	gossip.epochStore.Heads	(“” => eventId*)
v	gossip.epochStore.DagIndex (vecDb)	(for subtables bellow only)
vT	vecmt.Index.HighestBeforeTime	(eventId => time)
vS	vecfc.Index.HighestBeforeSeq	(eventId => seq)
vs	vecfc.Index.LowestAfterSeq	(eventId => seq)
vb	vecengine.EventBranch	(eventId => branchId)
vB	vecengine.BranchesInfo	(“c” => vecengine.BranchesInfo)
r	abft.Store.Roots	(frameId + validatorId + eventId => “”)
C	abft.Store.ConfirmedEvent	(eventId => frameId where confirmed)

Table 3.6: Caption epochDB gossip-epochId

3.2 Network performance bottlenecks

Table 3.7 shows processing time of new blocks in *Parity* implementation. Opera has exactly same problem since the way data is stored is identical. Opera is currently at 38 million blocks as amount of data in database rises the disk I/O operations take more time. This problem is further described in Experiments. 6.[1]

Blocks (M)	DB
0 – 1	261
1 – 2	1026
2 – 3	4719
3 – 4	4630
4 – 5	28617
5 – 6	54754
6 – 7	61769
7 – 8	72958

Table 3.7: Block processing time of DB (s).[1]

EVM -> FVM

Fantom virtual mashine would be way faster currently limit in EVM is set to 20 transactions per second and uses stack database - level db is being used (key value storage). FTM could start working with registries, but there would be new problems as concerns for security. EVM still in use so it is slow, will be replaced in future by FVM - fantom virtual machine.

Disk IO

Current disk write speeds are about 200 MB/s, while read speeds average at 300 MB/s with peaks at 800MB/s.

3.3 New node synchronization options

In this chapter I go trough currently available methods of synchronization in Fantom Opera network.

Synchronization from genesis

When opera is first run it is required to select genesis file. Depending on which genesis file is selected that network will be created. Next time opera is run it already has genesis data generated so it doesn't have to recreate it and continues from local highest head. When new node wants to connect to network and synchronize it has in configuration list of bootnodes. These bootnodes is list of entry nodes provided by the Fantom Foundation. Client adds bootnodes as peers and P2P communication is established. From its peers client appends his list of peers by active peers connected to his peers and so on.

Snapshot

This solution is based on implicitly trusting the datasource. Hosting server zips whole database. Data can be downloaded to new server in zip file for Fantom there is list available at *ultimatenodes* [3]. This is going to be the fastest solution as it fully utilizes network and disk and doesn't require any further processing power then unzipping into directory. Snapshots need to be updated regularly so if user wants to download database with most recent state

[3] `https://ftmbootstraps.ultimatenodes.io/`

it isn't out of sync for long time and new node doesn't have to compute huge number of passed blocks.

Regenesis

Synchronized server can create new genesis file, that has state of MPT archived and doesn't need previous events. There is same problem as with snapshots disadvantage of having to create newer and newer genesis files because synchronizing from older one would require catching up to head.

Chapter 4

Fast synchronization design

Catching up to head from genesis takes a lot of time. Fast sync is most non-DAG networks is easy because you only have to download newest block from network and you are quickly caught up up to head. But in DAG networks it gets trickier, because there isn't block as it own. So rather than downloading specific block you have to download set of events that will be needed catching up to head and possibly in future computations

4.1 Synchronization in other networks

On top of mentioned synchronization processes already available in Fantom Opera described in previous chapter 3.3 there are other ways of other blockchains. But not all are possible to be implemented in Opera due to the different network designs.

Non-Interactive Proofs of PoW

Non-Interactive Proofs of Proof of work (NIPoPoWs) are used for decentralized consensus protocols based on Proof of Work mining, where nodes are required to download data linearly from their current head. This concept enables synchronizing nodes to get the information of reached PoW consensus without sending all the information about it. Unlike traditional blockchain only generated `Superblocks` are sent. These superblocks are only logarithmically-sized proofs of blocks in specific time period. Since they are non-interactive they require only single message between prover and the verifier of the transactions. Clients synchronize with network quickly even if they remained offline for very large period of time. They are very useful in sidechains and lightweight clients, but not usable in DAG networks - since there would have to be important parent events in every node anyway. [9]

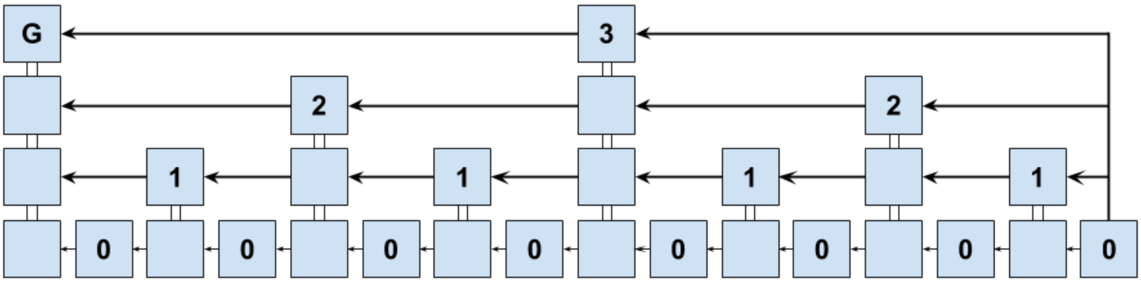

Figure 4.1: Confirmation by NIPoPoWs Superblocks between clients only every few blocks [2]

Full sync

Full sync is consider to be the most secure way of synchronization. It is the simplest solution, but the one that takes longest time. It is to let node synchronize by protocol itself from the genesis. Node downloads all events and computes them to receive full MPT tree with valid data. This method of classic synchronization is also used for nodes that were lagging and are trying to catch up. original way of propagating data to new nodes as if the node was lagging and trying to catch up. All transactions since beginning (or genesis) are being executed to catch up with the head. [16]

4.1.1 Fast sync

Fast sync is based on relying on Proof of Work mechanism. Not all transactions are executed, but only subset 64 blocks is downloaded. Tinkering with this many blocks would be computationally so expensive, so that this method is considered secore for construction of state root. By having state root, which node trusts it can download state trie directly (balances of account, states of contracts) because it can be confirmed by the HEAD state root. [16]

Warpsync

Every 5,000 blocks node takes snapshot of the current state. This snapshot can be then distributed to new nodes. Any node that synchronizes can download each individual chunk from any server that implements *warpsync*. Unfortunatelly where is no mechanism of how to verify transafer date in download progress - node has to finish download to start checking if all chunks arrived correctly. [19]

Snapsync

Snap sync is protocol to download data almost retrieved almost in real time. Synchronized nodes keep dynamic snapshots of recent states available for peers. Peers joing to the network download ranges of indexes with this data these ranges can be sumchecked by MPT as they are recieved. This prevents attacks for injecting poisoned records into database - compared to warp sync, where sumcheck can only be down afterwards. Snapshot in servers consists of

[2] Adopted from `https://nipopows.com/images/hierarchical-ledger.png`

only necessary data in flat storage, there it can be iterated trough in 7 minutes. Meanwhile iterating trough whole database on Ethereum can take up to 9,5 hours. [18]

4.2 Solutions overview

- **NIPoPoWs** - is designed for PoW
- **full sync** - downloads blocks and computes them one by one the to reach current state.
- **fast sync** - downloads all tree nodes from MPT tree
- **warp sync** - Used for useful data (accounts, storage slots), can be verified only after synchronization process (not during), so if any data gets corrupted by an attacker whole process has to be started over. Requires generated snapshot on hosting nodes before the download process.
- **snap sync** - user gets data depending on type of snapshot. Segments of data are being verified troughout the process by MPT hash check. Snapshot on hosting nodes is being generated in running node.

Comparisment

Comparisment of different blockchain's synchronization times. In Table 4.1 you can see how different concenses and their times of synchronization, size and TPS compare. The time of synchronization depends on the amount of data being synchronized. Out of this list the lightnode in Geth implemented by snap sync stands out as takes only **15 minutes** to be have verified state. (Note solana is more centralized.)

Blockchains stats	TTPS	Synchronization time	storage	bandwidth
Bitcoin full node (core)	7	7 days	350GB	5,5 GB day
Ethereum light node (geth)	30	15 minutes	500MB	25 MBit/s
Ethereum full node (geth)	30	3-4 days	500GB	25 MBit/s
Ethereum archive node (geth)	30	20-30 days	6TB	25 MBit/s
Fantom DAG	300 000	8 days	2.5TB	1250 MBit/s
Solana (2 day history) DAG	65 000	1 hour	1,5TB	300 MBit/s
Avalanche DAG	20000+	3 days	200GB	30 MBit/s
Cardano	1000	6 hours	30GB	10 MBit/s
Ethereum 2.0 L2	100 000	-	-	-

Table 4.1: Comparison of different blockchains synchronization speed

4.3 Picking new fast synchronization solution

First of all I needed to choose the way of synchronization and what amount of data should be synchronized.

Initially Proposed Design

New node asks peers not for events one by one, but using idea from to skip block computation. First of all it is important to get current list of validators, so we will trust the signatures of current blockchain state. This list could be recovered from ordinary P2P communication without adding another P2P communication interface. Client server will ask for data for each Epoch one by one since genesis file. It has to explicitly trust validator public keys that are written there. Every new epoch is signed by keys of previous validators and contains list of validators for following epoch. Once node catches up to last finished epoch we have list of currently valid validator public keys. Then events and state data (aBFT tree indexes from gossip db) in MPT need to be downloaded. The result would be node which is as big as an offline pruned node with all required data for further block computations.

Data needed for synchronization:

- **BlockState** - actualized after each block.
- **EpochState** - actualized every new epoch (5-10 mins on mainnet)
- **EpochEvents(E)** - list of events(DAG vertices) in epoch + indexes of aBFT algoritm
- **EVM's MPTs** - EVM'storage

At the start of epoch should (E) be empty. Therefore it makes most sense to synchronize database in exact point, where this list of events is empty.

Selecting from existing designs

Choosing mentioned in chapter 3.3 there is an option of snapsync, which was implemented and published to testnet at the time of me working on my original solution (described in 5.1). Then there are methods of synchronization by simply downloading prepared files by other users. The main problem is creation of these snapshots of database takes at hours aswel as prunning if the snapshot is meant to contain just recent data. Since most users don't need nor want historical data of blockchain I presumed these two options for them are covering most usecases. Then I looked how synchronization is taking place in company managing multiple servers. Since these servers are all considered secure in bounds of just few people having access to them. The synchronization process is considered safe when data of one server is transfered to new server. This is especially useful as the server will contain exactly the amount of history needed for usage (for example 1 month old data should be stored on all servers so they are easy to access etc.). Hence copying of these data by tools such as *rsync* [3] is optimal option. The biggest drawback of copying data directly from one server to another is that the hosting node has to be stopped. Even tho rsync is 7GB/min fast the downtime of server could have negative affect of service availability. For example if network would have 10 nodes and new 5 nodes for different service needed to be synchronized quickly then fastest way would be to stop 5 running servers and transfer data to each individual server. This would lower the service capacity.

4.4 Selected solution

My selected solution is to synchronize data from server without stopping the server itself and directly copy data from it. Since server isn't stopped it can maintain pace with network

[3] `https://linux.die.net/man/1/rsync`

and won't fall behind from current head. It is obvious that synchronization this way is going to be slower than *rsync* since processing power as well as disk IO is being used for computing new blocks in meantime. Also I have to keep in mind that with increasing size of database the write speeds are going to be slower.

Communication interface

Communication interface on figure 4.2 displays how two nodes will be sending and receiving data from one another. There will only be one hosting server connected to a client.

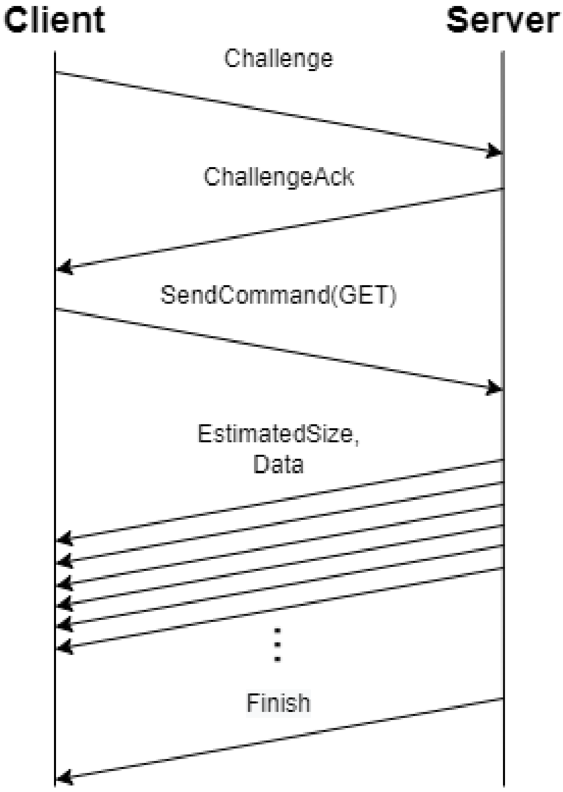

Figure 4.2: Comunication

- **Challenge** - Random number sent by client
- **ChallengeAck** - Signed value of **Challenge**
- **SendCommand(GET)** - command starting initializing download
- **EstimatedSize** - estimated size of database
- **Data** - contents of gossip database
- **Finish** - notification of end of a stream

Client

Client will connect to hosting server on given port. After establishing connection and receiving public key the data transfer will be done by pipelinning workload into three threads:

- **ReadBundle** - downloading bundles one by one and putting them into buffer for Hashing thread
- **Hashing** - reads data from previous thread and computes their hash and confirms it fits the given signature and correct server public key.
- **DbWrite** - Receives list of items from **Hashing** thread that were in bundles, which hashes and signatures were confirmed. Writes them as they arrive into database.

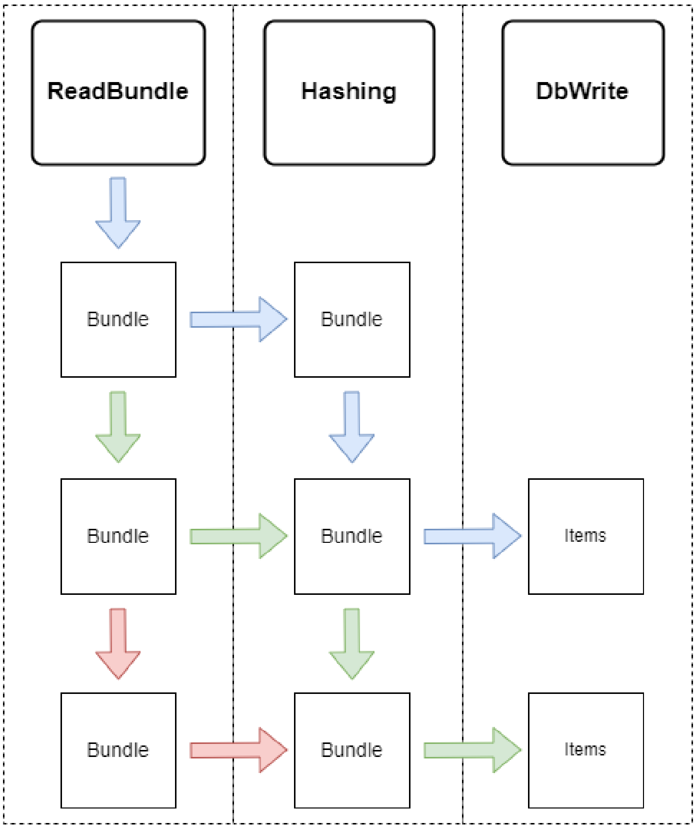

Figure 4.3: Client Data Pipelining

Server

Analogically to client. Server will wait on connection from client then sending him his public P2P key used used for transferred data verification. The process is also done by pipelining workload, but I decided to only use two threads. Data won't be avaiblale as fast as client reading them, but this should be enough because client will get stack inserting data into database:

- **DbRead** - Iterator reads data (Items) from database, groups them into bundled packages, to which hash and signatures are assigned.

- **SendBundle** - sends prepared bundles to client

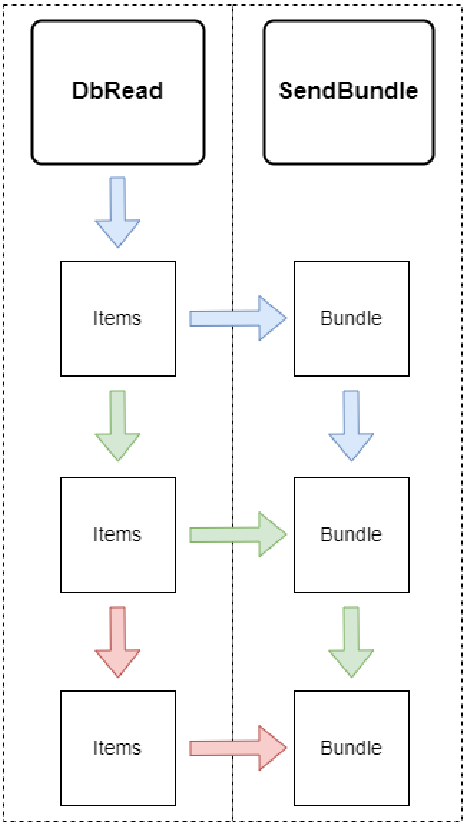

Figure 4.4: Server Data Pipelining

Chapter 5

Implementation

In this chapter I describe my implementation of fast synchronization in Fantom Opera network based on my designed solution described in previous chapter 4.4. First I describe what versions I build my solution on, then I go deeper into how I implemented communication protocol. After that I describe what else needed to be fixed to have clean state of database. Based on analysis of Code structure in 3.1 I understood that to run node I will need to have valid gossip database. In Lachesis database I will only need to have matching `dbFlushId`, epoch and validator list. Because proposed synchronization is based on state at end of epoch I don't need block event states, since they are only needed for currently ongoing epoch (and are blank at start of new epoch). The data in database is stored in RLP encoding, but since I'm transferring them in 1 to 1 ratio it I don't have to decode them.

5.1 Opera versions

I have implemented my solution in two Opera forks. I didn't fork directly from official repository[1]. There are many forks of go-opera, but I only found two with interesting features related to purpose of this thesis so I chose them.

Egor Lysenko's fork

- this version was particularly interesting to me, because as I was designing my solution version I found out that there is already version with *snapsync* taken over from Ethereum. Even tho this version is so far only available for testnet it doesn't make difference, when different amount of traffic and database size was taken into the consideration. Hence I used this version not only for doing experiments on my leveldb database, but also for comparison of how well his implementation performs. The specific version I picked is forked at commit `a491dc96fcef52557fd438d95843b5c1304badad` available on his github[2].

Honza Kalina's fork

The second version I picked is from my colleagues Honza Kalina. His version runs on mainnet. Has perk of having fixed issues such as dirty bit correction and has introduced

[1] `https://github.com/Fantom-foundation/go-opera`

[2] `https://github.com/uprendis/go-opera/tree/feature/customizable-genesis-file`

pebble database that I'm going to compare with *Leveldb*. The version I forked from his github[3] is from commit 36fcc55a71219a7bfa1578ee902c6e531e04547b.

5.2 Establisment of connection

As described in previous chapter in selected solution 4.4 portrayed in Figure 4.2, there is going to be key exchange from server to client before the sending of data begins. Client node generates random Challenge, server will sign it and return it in ChallengeAck. Public key is retrieved by crypto.Ecrecover(*hash, *signature) function in Golang *crypto* library implementation. This recovered key is written to program output right at the start so it is possible to double confirm that the server client is communicating with is in fact really the selected entity. Next step is for client to send request to get data. In the implementation there is simply "get" message sent to the server. Server responds with database size estimate read from selected datadir in /chaindata/gossip subdirectory. Then the progress of data transfer begins.

Data lifecycle

The data transferring package in my implementation is called Bundle - it is package of multiple items compounded together. When data length of items surpasses the as they reach minimal recommended size - RECOMMENDED_MIN_BUNDLE_SIZE, then hash and signature are computed to it and bundle is sent. Bundle consists of:

- Finished (bool) - flag determining that stream has successfully finished.
- Hash ([]byte) - Keccak256Hash of data.
- Signature ([]byte) - hash is signed by secp256k1 - EDCSA private key used in P2P communications
- Data ([]Item) - item is structure composed just by Key ([]byte) and Value ([]byte)

On each composed bundle RLP encoding[4] is used. Is used to efficiently encode and decode between any data structure and []byte. This stream of bytes is then put into *lz4* compression and RLP encoding again. This last RLP enconing at the server side is important so the client side knows how long is the incoming compressed message. Data is sent by TCP socket to the client side. Client is using RLP decode on incoming bytes as they arrive and tries to decode structure. The first structure for every bundle it decodes is []byte containing compressed data. *Lz4* decompression is called and []byte containing encoded by *RLP* is retreived. After RLP decoding. Data is hashed and by Signature the hash is verified. The last step is to put data into database. See Figure 5.1 for data path at client side and Figure 5.2 for server side.

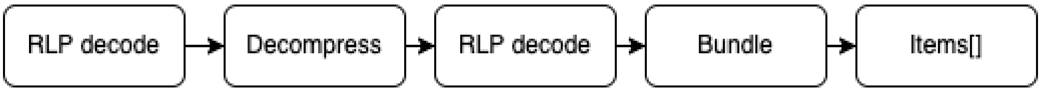

Figure 5.1: Client: data being processed from network to the database

[3] https://github.com/hkalina/go-opera/commits/pebble-v1.1.0-rc.5

[4] https://eth.wiki/fundamentals/rlp

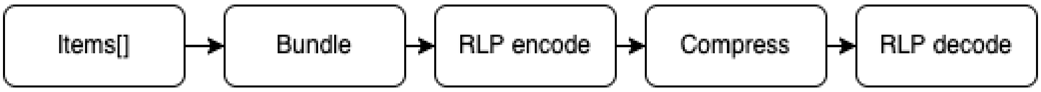

Figure 5.2: Server: data being processed from database to the network

5.3 Hosting server

Hosting server is implemented in `direct_sync/sync_server.go`. As Opera is launched new thread is started for handling client requests. Client is started by function `InitServer(gossipPath string, key *ecdsa.PrivateKey)` with parameters for readding estimated size of database and `secp256k1 - private key` used for signing onto hashes and challenges. As server is running and new epochs are computed reference to `gossip.SnapshotOfLastEpoch` is kept. I implemented this snapshotting in a way that whenever block is being committed, there is check if it isn't considered to be end of epoch. If it is the previous snapshot gets overwritten. But client can't download data from server without snapshot - so client might need to wait few minutes for new epoch to start. I also implemented basic mechanism of keeping track of how many clients are connected. This isn't fully viable since I ended up enabling only one specific port for this tool. As it isn't expected to be used for multiple servers at once. The following Algorithm 1 describes how bundles are being sent.

Algorithm 1 Server side sending bundles

```
function DBREAD(sendingQueue)
    while $Iterator.hasNext() \neq nil$ do
        $item \leftarrow Iterator.next()$                          ▷ Reading data from database
        $append(Bundle.Items, Item)$                              ▷ Grouping into bundles
        if $Bundle.size \geq RECOMMENDED\_MIN\_BUNDLE\_SIZE$ then
            $bundle.hash \leftarrow hash(bundle.Items)$           ▷ inserting hash
            $bundle.signature \leftarrow signature(bundle.Items)$ ▷ inserting signature
            $sendingQueue \leftarrow bundle$                      ▷ bundle to sendingservice thread
    if $Bundle.size > 0$ then                                     ▷ Send last items
        $bundle.hash \leftarrow hash(bundle.Items)$
        $bundle.signature \leftarrow signature(bundle.Items)$
        $sendingQueue \leftarrow bundle$
    close(sendingQueue)
function SENDINGSERVICE(sendingQueue)
    while $sendingQueue.isOpen() = True$ do
        $bundle \leftarrow sendingQueue$
        $socket \leftarrow bundle$                                ▷ Send bundles to client
```

5.4 Client server

Client implementation starts in `cmd/opera/launcher/launcher.go`. First of all the chaindata dir with gossip database are prepared. Data synchronization from server is in `direct_sync/sync_client.go` file. When connection is established client receives public key of server and uses this key to confirm data it is downloading for detailed description about connection read 5.2 section. When data download in client is finished the process continues by creating Lachesis database. Lachesis needs to contain epoch number and list of current Validators. The last thing that needed to be done was to insert same FlushIDKey into both databases with value of current time. This process is visualized in Alg. 2 and 3

Algorithm 2 Client side receiving bundles

```
function ReadBundle(hashingQueue)
    while ($bundle \leftarrow socket$) $\neq nil$ do                ▷ Reading bundles from socket
        $hashingQueue \leftarrow bundle$                            ▷ Send to hashing thread
    $close(hashingQueue)$
function hashing(hashingQueue, writeQueue)
    while $hashingQueue.isOpen() = True$ do
        $hash \leftarrow hash(bundle.Items)$                         ▷ Calculate hash
        $publicKey \leftarrow signature(bundle.Items, hash)$         ▷ Calculate publicKey
        if $hash = bundle.Hash$ & $publicKey = bundle.Signature$ & $publicKey = presharedKey$ then   ▷ Verify data
            $DbWrite \leftarrow bundle.Items$                        ▷ Send to writting thread
    $close(writeQueue)$
function DbWrite(writeQueue)
    while $writeQueue.isOpen() = True$ do
        $db \leftarrow bundle.Items$                                 ▷ Insert into database
```

Algorithm 3 Database correction

```
function correctDatabase(gossip, lachesis)
    $gossip.FlushIDKey \leftarrow time.Now()$                        ▷ Insert new flushIdKey
    $lachesis.epoch \leftarrow gossip.epoch$ CommentUpdate epoch
    $lachesis.validators \leftarrow gossip.validators$               ▷ Update list of validators
    $lachesis.FlushIDKey \leftarrow gossip.FlushIDKey$               ▷ Setting same flushIdKey
```

Chapter 6

Experiments

For gathering the following metrics I used Prometheus [1] tool built into go-opera in combination with Grafana[2] to visualize the collected data. To each experiment running modified version of opera I enlisted linked branch with versions of code the experiment ran. Testing was composed on two servers with exactly same configuration 6.1.

processor	AMD's Ryzen™ 9 5950X on Zen 3 architecture, 16 cores, 32 threads
memory	128GB DDR4 ECC RAM
storage	2x 3.84 TB NVMe SSD Datacenter Edition (unspecified manufacturer) ZFS RAID 0 (Striping) on data folder
network	1 GBit/s guaranteed bandwidth

Table 6.1: Specifications of both servers running experiments.

Time of block processing

Block processing time depends on current server load as well as the difficulty of the processed block. In Figure 6.2 I portrayed how block processing on server is delayed when synchronization is enabled. In the first part of the image you can see server takes only few milliseconds to process new blocks. The first part of the image server actually isn't synchronized to head and is catching up only right after it synchronized it starts the direct sync server hosting and client connected and synchronization began. Even tho time rises dramatically server still keeps on with head. To compare in Fig. 6.1 you can see how much is server delayed when RPC server processing 100 requests per second is running on it. Even tho server is synchronized the processing of block takes a lot more time, but still it isn't a problem. Yellow line is color taken from metric `chain/execution` and the green line stands for values in `chain/inserts`. From this graph we can see that even two processing time fluctuates database is more or less keeping up with newly incoming and processed blocks.

[1] https://prometheus.io/docs/guides/go-application/

[2] https://grafana.com/products/enterprise/

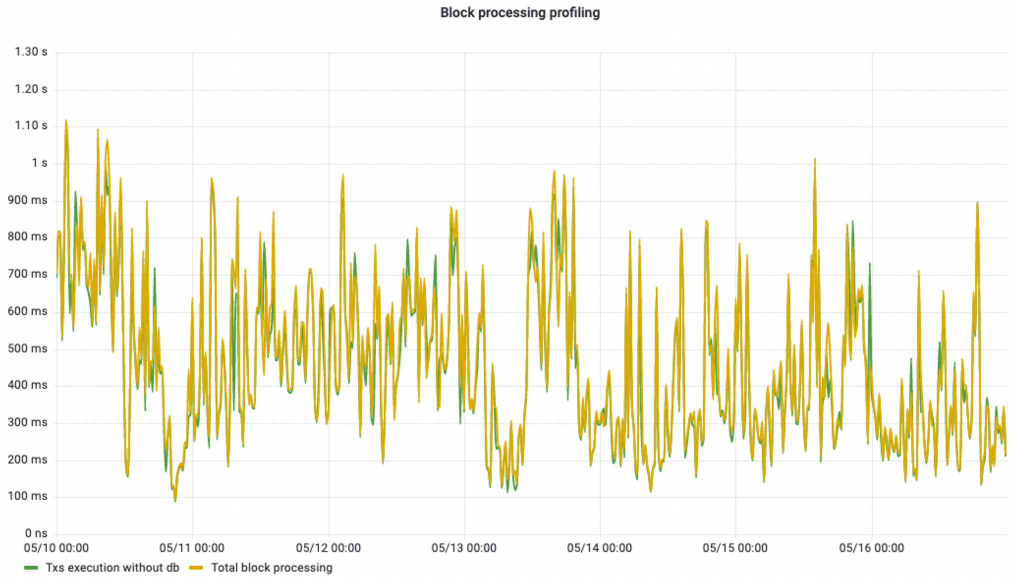

Figure 6.1: Block processing time as node is under load of RPC requests

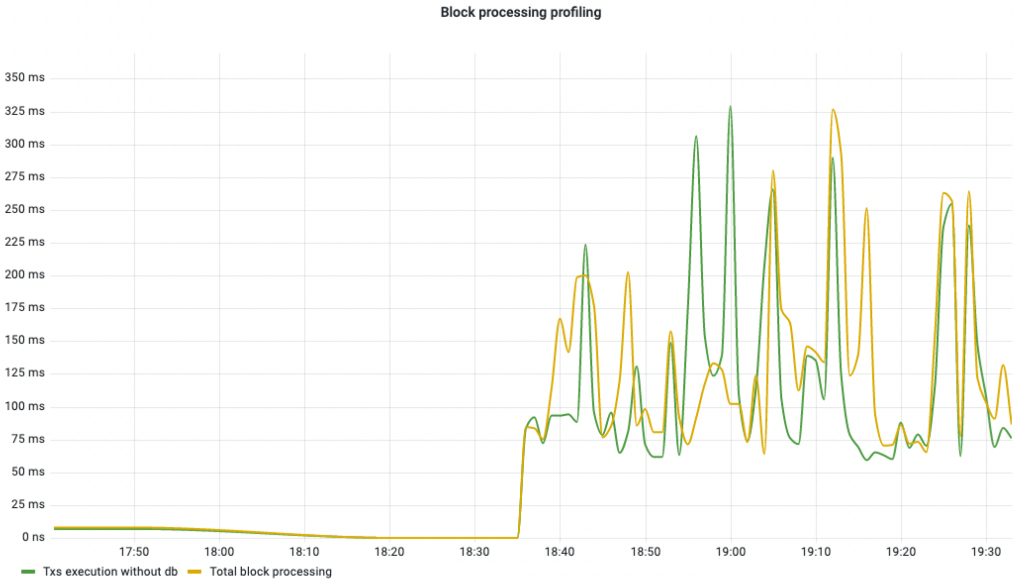

Figure 6.2: Block processing time when direct sync is launched

We can expect that in database with size of 800 GB the block processing time in the best case scenario - when there isn't any other job taking up disk or processor time. The speed of block importing trough P2P network is about 2000 blocks per minute. To compare with the speed of new blocks being emitted at 10 transactions per second (TPS), then there are about 2500 blocks an hour generate. The number of new blocks fluctuates a lot sometimes on mainnet it can be as get as low as few hundreds of new block in hour. But if there aren't many transactions in chain the final catching up process could in ideal conditions matter of few tens minutes for synchronization of one day data.

Performance measurements

In this section I compare different synchronization types as well as different metrics, which play role in synchronization time. For initial data I prepared data of 750GB pebble db (pruned history), synchronized leveldb by snapsync 85 GB synchronized with full mpt state [3] and prepared snapshot state[4].

Test	Database	Time	Speed
Full synchronization from genesis	pebble	~8 days	**2870672 blocks/day**
Full synchronization from genesis	leveldb	~12 days	**5235422 blocks/day**
snapsync (testnet-6226-pruned-mpt.g)	leveldb	**17 mins**	**0.6 GB/min**
snapsync (testnet-6226-full-mpt.g)	leveldb	**41 mins**	**2.859 GB/min**
snapshot	-	-	network limit

Table 6.2: Comparison of how synchronization times

- **genesis pebble** - it took 10.5 minutes to decode genesis.
- **genesis leveldb** - it took 5 minutes to decode genesis.
- **testnet-6226-pruned-mpt.g** - 2.99 GB genesis file downloaded in 30 seconds then synchronization itself took 16.5 mins and size of database in filesystem was 10GB.
- **testnet-6226-full-mpt.g** - 71.23 GB downloaded in 11 mins with 30 min it too to synchronize database 85GB. synchronized downloading the whole MPT occurs at snapshots after synchronization the head was 3 hours and 10 minutes old.
- **snapshot** - database compressed into zip

Bottlenecks

In Figures 6.3 and 6.4 you can see how bottlenecks during the transfer look like.

[3] `https://github.com/Fantom-foundation/lachesis_launch/blob/feature/add-snapsync-instruction/docs/genesis-files.md`

[4] https://docs.fantom.foundation/node/snapshot-download

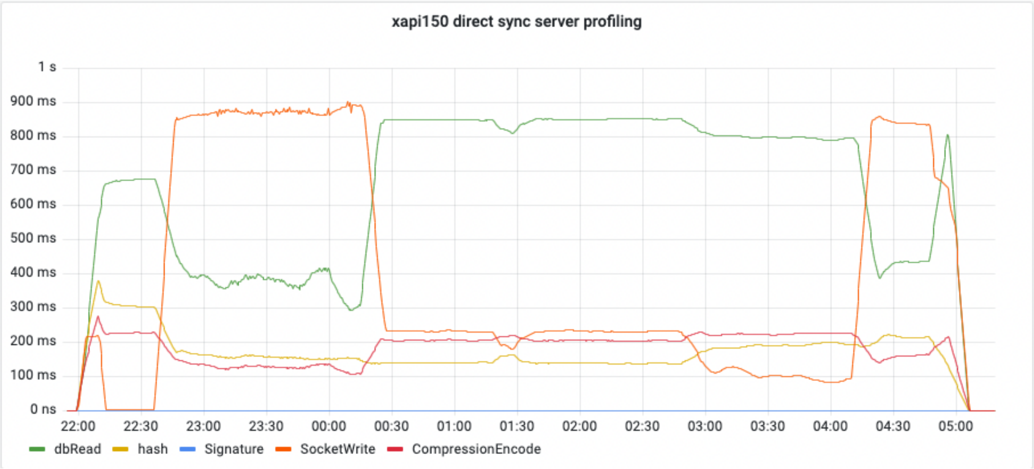

Figure 6.3: Server side bottlenecks

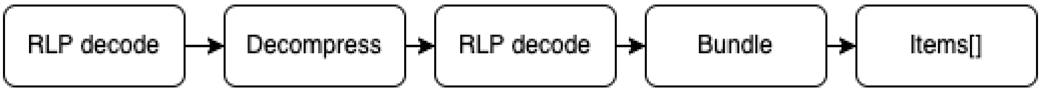

Figure 6.4: Client side bottlenecks

Pebble db fix

Pebble DB was at first really slow the process of data transferring for 750GB took over 36 hours. The problem was old in pebble version. At the end of progress on download I was measuring the statistics of database by command `mainDB.Stat(„metrics")`[5]. I found out that the old version did compact on it's own 35012 times and data in compact queue had size of 1.5 TB. the number of compacts was almost 10 times bigger 3051881 and the compact queue had 232 GB size. I tried to fix this on my own by explicitly calling compact every 5 MB - the time of transfer dropped from 36 hours to 9,5h hours. But upgrading the pebble version (from `v0.0.0-20220314154659-f9d4a33d7897` to new veresion at `v0.0.0-20220517003944-e567fec84c6e`)and letting it do compact on its own was even more efficient and the final time got to 7 hours.

RLP encoding

I had to checked, whether the `RLP` encoding isn't increasing the data size too much. The increased size of all data items was increased.

```
actualKeyValueData = 43,073,229,795
rlpEncodedData = 44,013,463,068
```

[5] `https://github.com/cockroachdb/pebble/blob/master/metrics.go`

Lz4 compression

The decompression with `Lz4` worked really well. The total Amount of data was transfered was reduced from 1484.1GB GB to 751.2GB (in database with size of 850.1GB). See Figure 6.5 for the graph of how well the compression was doing during the process.

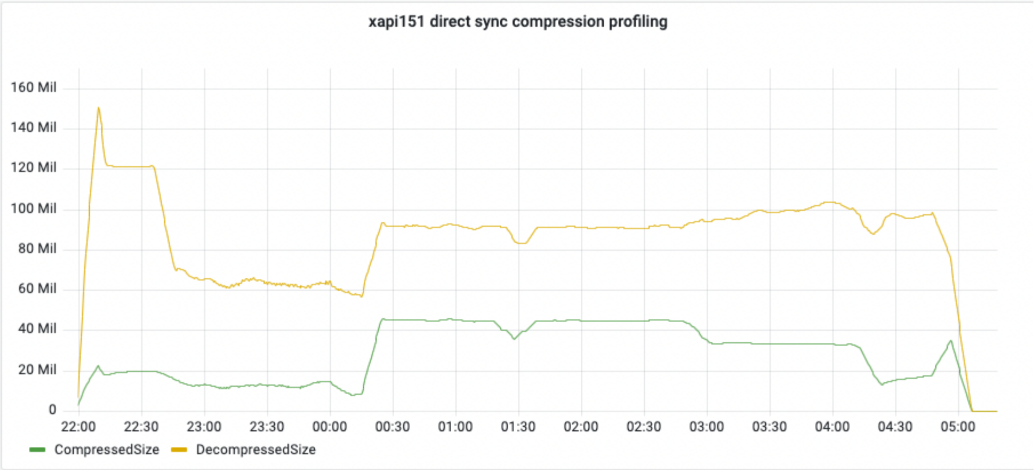

Figure 6.5: Visualization of transferred information compared to lz4 compressed size

My implementations results

In Table 6.3 you can see how well each of my branches performed.

Branch	DB size	time
`leveldb-direct-sync-buffer-1-compression`	**88GB**	**20min**
`pebble-direct-sync-old-pebble`	**779GB**	**36h44min**
`pebble-direct-sync-old-pebble-compact`	**822GB**	**9h21min**
`pebble-direct-sync-buffer-100-compression-batch`	**897GB**	**7h14min**
`pebble-direct-sync-buffer-1`	**838GB**	**7h**
`pebble-direct-sync-buffer-1-compression`	**850GB**	**6h58min**

Table 6.3: Block processing time of DB (s).

6.1 Efficiency evaluation and further improvements

The data stored in *pebble* as well as *leveldb* ended up being way more compressed then I anticipated. The measured data in database stored in database with 759 GB disk size had keys and values with total amount of 1372.26GB. I managed to lower the amount of traffic bandwidth required introducing compression, which ended up being about 50% efficient. Unfortunately from data I found out that server is big chunk of time slowing down whole process of synchronization. The read speed from database falls when data read from database is read in smaller sizes. But there is also time when process is waiting on client to write data into database. The write speed at client is not able to be improved a lot. I tried different methods of increasing the process from creating `batch` writes, to increasing buffers for more efficient pipe lining, but the process of synchronization peaked at 2,8GB/min with database of size round 800GB. Writing speed is increasing as the size of database increases. In comparison when database had 86 GB the my solution was 4,2 GB/min fast, which when compared to `rsync` speed at 7GB/min is more acceptable. As a further improvement I would like to mention the possibility of downloading state from multiple servers at once. This implementation would require more sophisticated mechanism of retrieving data from server and would need to be thoroughly verified, since the data on multiple servers are not identical. I didn't end up having enough time for testing leveldb as first iterations of pebble took me around over 30 hours run time until I fixed the version bug. But the comparisment between speed of `leveldb` and pebble ended up being comparable in smaller scaled writes - but in bigger writes I presume that `leveldb` would be doing much worse as the speed of block processing since genesis was almost twice as long.

6.2 Security summary

State of synchronized blockchain can be verified by stopping it and recalculating every transaction from genesis. This takes over 1 month with 2 TB of data. So when we download any foreign data we might have incorrect header. To include transactions in history is practically impossible since there are continuous check sums in MPT, but if attacker includes malicious transactions in history, recalculates fork to current head the fork could be mistaken for the original since node doesn't know what to compare it with. Of course then bootnodes and peer nodes available to attacked server would have to be faked running on the same fork with wrong data.

My solution is based on client implicitly trusting server when downloading data, preferably having pre-shared public P2P key of the server. When program is launched and downloading starts public key is retrieved from the given challenge and client can verify it.

By using this key all hashes of bundles of items are signed and the signatures are attached to bundles as well. Client side before every write to database makes sure the signature corresponds the hash of received files. Therefore received data can't be exchange during the process. The transfer could be disrupted by sending incorrect data to client or by DoS attack on hosting opera node, draining all it's resources. So extra attention has to be payed on settings of the firewall rules. But the final result of synchronization should be same as the data of the server at the time of snapshot creation.

Chapter 7

Conclusion

The goal of this thesis was to get familiar with the existing methods of synchronization of new nodes into existing blockchain protocols, mainly in the Fantom-Opera. Then, by studying current state of node synchronization, I have realized that there isn't any decently fast solution as in other networks. While working on this thesis. I have learned a lot about their different approaches and synchronization processes. Synchronization in *Fantom-Opera* wasn't improved on for a long time because the number of people wanting to run nodes was still pretty small and there wasn't a big demand for it. However, about year ago, Fantom was on a rise and the amount of running nodes rose. Since Fantom is a DAG PoS based blockchain, the data structure is bit more complicated to synchronize than PoW like Bitcoin/Ethereum with the need for only a few newest blocks to have a valid state.

I implemented my designed solution and experimented with different modifications to improve it. The final selected solution consists of client and server - data is directly transferred, with hashing and signature verification, so data tinkering by an attacker should be impossible. You can see these results in the chapter Experiments 6. My solution is slower than `rsync` as expected to be in the best case scenario possible. The final solution's speed, unfortunately, isn't as good as I hoped for. The main factor is that the increasing size of the database gradually increases write time of the client - so when the synchronized database is only in 10-50 GB transfer speed, it is only a few percent slower than `rsync`, but when the database is at 800 GB, the average transfer speed decreases and is at about 30% of the hypothetical transfer speed. I didn't expect writing to the database to get this slow. Another interesting finding was that the RLP encoding of data with headers increased the given size only by few percent, while with pebble/leveldb the compression rate of about 50% of the size of the data on the disk compared to the actual size of items. This rise of additionally needed bandwidth was mitigated by using *lz4* compressing that halved the size of data needed to be transferred. However, there was a bottleneck at the server side as well. There were segments of data, which were sent faster by the server, that writing sped and vice versa actually about 50% of the transfer time. This was caused by various data sizes at the different prefixes. Since iterator goes through the database by prefix that, if it starts sending prefix with smaller values, then the data processing and loading speed decreases. As I worked, I still kept in mind that the hosting server shouldn't experience too much of a load. I succeeded with a given configuration of servers by confirming that I'm unable to slow down the hosting server for it to start lagging. This was confirmed by looking at the current block and processing time of new blocks, while a client was downloading data.

Bibliography

[1] Baird, K., Jeong, S., Kim, Y., Burgstaller, B. and Scholz, B. *The Economics of Smart Contracts.* arXiv, 2019. DOI: 10.48550/ARXIV.1910.11143. Available at: `https://arxiv.org/abs/1910.11143`.

[2] Bowden, R., Keeler, H. P., Krzesinski, A. E. and Taylor, P. G. *Block arrivals in the Bitcoin blockchain.* arXiv, 2018. DOI: 10.48550/ARXIV.1801.07447. Available at: `https://arxiv.org/abs/1801.07447`.

[3] Choi, S.-M., Park, J., Nguyen, Q. and Cronje, A. *Fantom: A scalable framework for asynchronous distributed systems.* arXiv, 2018. DOI: 10.48550/ARXIV.1810.10360. Available at: `https://arxiv.org/abs/1810.10360`.

[4] Connor, R. O. Simplicity. In: *Proceedings of the 2017 Workshop on Programming Languages and Analysis for Security.* ACM, Oct 2017. DOI: 10.1145/3139337.3139340. Available at: `https://doi.org/10.1145%2F3139337.3139340`.

[5] Homoliak, I., Venugopalan, S., Reijsbergen, D., Hum, Q., Schumi, R. et al. The Security Reference Architecture for Blockchains: Toward a Standardized Model for Studying Vulnerabilities, Threats, and Defenses. *IEEE Communications Surveys Tutorials.* 2021, vol. 23, no. 1, p. 341–390. DOI: 10.1109/COMST.2020.3033665.

[6] Jeltsch, W. *A Process Calculus for Formally Verifying Blockchain Consensus Protocols.* arXiv, 2019. DOI: 10.48550/ARXIV.1911.08033. Available at: `https://arxiv.org/abs/1911.08033`.

[7] Kaleem, M., Mavridou, A. and Laszka, A. *Vyper: A Security Comparison with Solidity Based on Common Vulnerabilities.* arXiv, 2020. DOI: 10.48550/ARXIV.2003.07435. Available at: `https://arxiv.org/abs/2003.07435`.

[8] Karaarslan, E. and Konacakli, E. Data Storage in the Decentralized World: Blockchain and Derivatives. In:. December 2020, p. 37–69. DOI: 10.26650/B/ET06.2020.011.03. ISBN 978-605-07-0743-4.

[9] Kiayias, A., Miller, A. and Zindros, D. Non-interactive Proofs of Proof-of-Work. In: Bonneau, J. and Heninger, N., ed. *Financial Cryptography and Data Security.* Cham: Springer International Publishing, 2020, p. 505–522. ISBN 978-3-030-51280-4.

[10] Lamport, L. *Different Consensus Algorithms* [online]. [cit. 2022-01-10]. Available at: `https://lamport.azurewebsites.net/pubs/time-clocks.pdf`.

[11] Nakamoto, S. *Bitcoin: A Peer-to-Peer Electronic Cash System.* 2009.

[12] Nguyen, Q. and Cronje, A. *ONLAY: Online Layering for scalable asynchronous BFT system.* arXiv, 2019. DOI: 10.48550/ARXIV.1905.04867. Available at: https://arxiv.org/abs/1905.04867.

[13] Nguyen, Q., Cronje, A., Kong, M., Kampa, A. and Samman, G. *StakeDag: Stake-based Consensus For Scalable Trustless Systems.* arXiv, 2019. DOI: 10.48550/ARXIV.1907.03655. Available at: https://arxiv.org/abs/1907.03655.

[14] Nguyen, Q., Cronje, A., Kong, M., Lysenko, E. and Guzev, A. *Lachesis: Scalable Asynchronous BFT on DAG Streams.* arXiv, 2021. DOI: 10.48550/ARXIV.2108.01900. Available at: https://arxiv.org/abs/2108.01900.

[15] Niu, J. and Feng, C. Selfish Mining in Ethereum. arXiv. 2019. DOI: 10.48550/ARXIV.1901.04620. Available at: https://arxiv.org/abs/1901.04620.

[16] Szilágyi, P. *Geth v1.10.0* [online]. [cit. 2022-02-20]. Available at: https://blog.ethereum.org/2021/03/03/geth-v1-10-0/.

[17] team, E. *Ethereum Improvement Proposals* [online]. [cit. 2022-02-15]. Available at: https://eips.ethereum.org/.

[18] team, E. *Ethereum Snapshot Protocol (SNAP)* [online]. [cit. 2021-12-5]. Available at: https://github.com/ethereum/devp2p/blob/master/caps/snap.md.

[19] team, E. *Warp Sync Snapshot Format - Wiki* [online]. OpenEthereum Documentation [cit. 2021-2-5]. Available at: https://openethereum.github.io/Warp-Sync.

[20] Truby, J., Brown, R. D., Dahdal, A. and Ibrahim, I. Blockchain, climate damage, and death: Policy interventions to reduce the carbon emissions, mortality, and net-zero implications of non-fungible tokens and Bitcoin. *Energy Research & Social Science.* 2022, vol. 88, p. 102499. DOI: https://doi.org/10.1016/j.erss.2022.102499. ISSN 2214-6296. Available at: https://www.sciencedirect.com/science/article/pii/S221462962200007X.

[21] Wang, C., Chu, X. and Yang, Q. *Measurement and Analysis of the Bitcoin Networks: A View from Mining Pools.* arXiv, 2019. DOI: 10.48550/ARXIV.1902.07549. Available at: https://arxiv.org/abs/1902.07549.

[22] Wang, W., Hoang, D. T., Hu, P., Xiong, Z., Niyato, D. et al. A Survey on Consensus Mechanisms and Mining Strategy Management in Blockchain Networks. *IEEE Access.* Institute of Electrical and Electronics Engineers (IEEE). 2019, vol. 7, p. 22328–22370. DOI: 10.1109/access.2019.2896108. Available at: https://doi.org/10.1109%2Faccess.2019.2896108.

[23] Werner, S., Pritz, P. and Perez, D. *Step on the Gas? A Better Approach for Recommending the Ethereum Gas Price.* March 2020.

[24] Yaga, D., Mell, P., Roby, N. and Scarfone, K. *Blockchain technology overview.* Oct 2018. Available at: https://doi.org/10.6028%2Fnist.ir.8202.

[25] Zhang, H., Jin, C. and Cui, H. A Method to Predict the Performance and Storage of Executing Contract for Ethereum Consortium-Blockchain. In:. June 2018, p. 63–74. DOI: 10.1007/978-3-319-94478-4_5. ISBN 978-3-319-94477-7.

Appendix A

Contents of the included storage media

- `\README.txt` - instructions
- `\go-opera-leveldb.diff` - changes to leveldb version
- `\go-opera-pebble.diff` - changes to pebble version
- `\go-opera-leveldb` - direct sync in version forked from Egor Lysenko repository [1]
- `\go-opera-pebble` - direct sync in version forked from Honza Kalina repository [2]
- `\Thesis` - latex source code and images for this thesis
- `\thesis.pdf` - thesis text in pdf format

[1] https://github.com/uprendis/go-opera/commit/138befdd262aa584ce7a71871110dbafd10dbc6a

[2] https://github.com/hkalina/go-opera/commit/c644bee01a4d7dd1f775f1bc7ad65e90d2830b5f

Appendix B

Manual

For loading go-opera with direct sync to your device you can use attached CD disk or preferably u can download the newest version from my github[1]. You will need two nodes running side by side. On your server node start Opera with `datadir` which has path to Opera data, but you will also need to add flag `directsyncserver`. Note that if you don't have server with data you can choose between available options of synchronization 3.3.

```
./opera --directsyncserver --datadir ~/.opera/mainnet
```

On client node side you have to set `directsyncclient` with value of hosting server address.

```
./opera --directsyncclient=8.8.8.8. --datadir ~/.opera/mainnet
```

Once client is connected it will print out public key of server node and check snapshot for download is ready. If snapshot is ready then download starts and once it finishes it client node automatically launches computes new block in blockchain.

[1] `https://github.com/matejmlejnek/go-opera/tree/pebble-v1.1.0-rc.5-fixdirty-direct-sync-buffer-1-compression`