Filecoin Consensus

Performance analysis

Master Thesis

Marcel Würsten

Faculty of Science
at the University of Bern

September 2022

Prof. Dr. Christian Cachin

Cryptology and Data Security Group
Institute of Computer Science

University of Bern, Switzerland

Abstract

Filecoin is a public, open-source cryptocurrency designed as a decentralized storage network with the vision to store humanity's most important information. While the total storage capacity and the amount of miners steadily increases, Protocol Labs the inventor and maintainer of Filecoin have limited information on what limits the Filecoin consensus algorithms performance and with that what needs to be improved in further updates, since it is challenging to measure. This thesis is motivated by this lack of knowledge to build a foundation to test the Consensus Algorithm in a test network and explore the limits. We found the saturation point, where the throughput does not grow with more load for different networks sizes as well as different network topologies.

Contents

1 Introduction

Since Satoshi Nakamoto started the first blockchain in 2008/2009 as the public ledger for the cryptocurrency bitcoin [23], making it the first cryptocurrency to solve the double-spending problem without a central server or trusted authority, countless blockchains have emerged. Over time many blockchains were created, trying to avoid Bitcoins wasteful Proof of Work (POW) and instead implement a different consensus algorithm. Any block that is added to the blockchain by the Bitcoin network must demonstrate its legitimacy using the PoW consensus process. Since the task calls for a sizable amount of processing power, miners are incentivized to complete it honestly and are not rewarded for performing subpar work. Each node independently verifies the transactions, and any incorrect transactions included in the block are discarded as illegitimate. Because there are so many miners, it is extremely difficult for one entity to control 51% of the miners, a sort of sybil attack known as the 51% attack, which due to the power requirement is essentially impossible in the bitcoin network.

Proof of Stake (POS) used for example by Peercoin [8], Algorant [1] or Cardano [6] is an example for a different consensus algorithm. POS is based on owning a stake in the network and the creator of a new block is chosen randomly depending on its stake. Since the first white paper of Filecoin in 2014 [4] from Juan Benet contained the idea of a cryptocurrency operated file storage network, a few things have changed since then. One thing that stayed is the vision that Bitcoin's proof-of-work is wasteful and there needs to be a proof-of-work which does at least some useful work. The idea of Filecoin is that providing disk storage to a decentralized storage network is the required work. As a motivation for a lot of people to join and work for the network, Filecoin creates a strong monetary incentive because one gets rewarded for provided storage. While the paper has some ideas around how such a system might be implemented, apart from the basic idea, most things changed or evolved in the process of implementing the system. With all the hurdle of creating such a system, Protocol Labs launched a public Filecoin network in autumn 2020 [20] and continues the development of the system. As of writing these lines, most of the Filecoin protocol specification topics have reached the "Reliable" state and the theory audit begun [21]. The Filecoin team has engaged recognized third-party auditing experts to make sure that the protocol's design and implementation actually serve their intended goal of making Filecoin a safe and secure network.

ConsensusLab as one of the research groups within Protocol Labs is exploring the implementation of

new consensus algorithm and protocols with the goal of improving the Filecoin network performance. But in order to test their prototypes, they need a testbed to benchmark Filecoin's performance.

Thus the goal of this thesis was to build a testbed to analyze the performance of Filecoin's Expected Consensus algorithm, with ideally the possibility to leverage ConsensusLab and other teams to run performance benchmark with basic configurability over different network setups such as network size.

The theoretical background of the most important concept as well as of the used technologies are covered in Chapter 2. Chapter 3 offers explicit details of the Filecoin specification, as well as the design of our Filecoin testbench. A brief overview over the gathered results is given in Chapter 4 and the conclusion and discussion about future work can be found in Chapter 5.

2 Background

2.1 Blockchain

A blockchain is defined as a growing list of records that are cryptographically linked, typically using a cryptographic hash of the previous record, a timestamp, as well as the record data. Since each block references the hash of the previous block, a blockchain is resistant to data modification since any data cannot be altered without also altering all the following records. Typically blockchains are used in distributed peer-to-peer networks as publicly distributed ledger, where they solve the double-spending problem without the need of a central server or trusted authority. Blockchains can either be permissioned also called private i.e. you can't join unless you are invited to do so or permissionless also called public i.e. anyone with internet access can join as participant or validator [24]. One of the largest and most known public blockchain is the Bitcoin blockchain of Satoshi Nakamoto as being the first one to solve the double-spending problem for cryptocurrencies without a central server or trusted authority [23].

2.1.1 Filecoin

Filecoin is a decentralized storage network providing a cloud storage market based on a blockchain with its own token also called "Filecoin" or short "FIL", in which miners earn tokens by providing storage to clients. Hence, clients spend FIL to hire miners to store their data. Miners compete to mine block by providing storage, where the reward is directly proportional to the overall network storage. With that, unlike other blockchains such as Bitcoin, miners provide a useful service to clients besides just maintaining blockchain consensus. This creates the incentive for miners to provide as much storage as they can. The protocol then clusters these resources into a self-healing storage network based on the InterPlanetary File System (IPFS). Self-healing therefore, because the Filecoin blockchain can detect defective storage nodes and redistributes their stored files to reliable nodes. Thus Filecoin can be viewed as an inventive layer on top of IPFS [27][5].

The Filecoin blockchain system is composed from multiple different components and subsystems. The storage subsystem for example ensures one can participate in the Filecoin storage market by participating in storage deals and store/retrieve client data. It also runs a "Storage Power Consensus" to agree on the current state of the system. The power table ensures that the amount of blocks a given miner generates

through leader election is proportional to their provided amount of storage over the same period. The power table is adjusted for new storage commitments (increasing), expiring sectors (decreasing), terminated sectors (decreasing) or of a miner failed to provide the required proofs (decreasing) [11][21].

2.2 Consensus

In a distributed system the fundamental problem is to achieve an overall reliable system in the presence of a number of faulty components. This requires that the participants reach consensus over some data value needed for the computations. For distributed blockchains, this means that if a block is added to the blockchain all the nodes of the blockchain agree upon that. [25].

Probably the most famous consensus algorithm is Nakamoto Consensus i.e. Proof of Work (PoW) as sybil resistance and selecting the longest chain as the valid one. In PoW a miner is selected to create the next block based on who can solve a computational puzzle that requires a lot of computational power the fastest. Bitcoin is one example of a blockchain that uses PoW as its consensus algorithm. Alternatively to PoW one could use Proof of Stake (PoS). Instead of investing in expensive hardware to solve a complex puzzle, so called validators invest in the coins of the system by locking up some of their coins as stake. This stake either then acts as collateral that can be destroyed if the validator behaves dishonestly, because validators are responsible for checking that new blocks propagated over the network are valid and occasionally are allowed to create and propagate a new block themselves. One of the main benefits of PoS compared to PoW is the better energy efficiency as there is no need for wasteful work to be done just to solve a computational puzzle [17]. But there are also numerous other consensus algorithm, with a large family of them based on the Byzantine Fault Tolerance problem [7].

2.2.1 Expected Consensus

Filecoin uses a so-called “Expected Consensus” or short “EC” algorithm for consensus in the distributed network. EC is a probabilistic Byzantine fault-tolerant consensus protocol that operates by running a secret leader election every epoch, in which by expectation, a set number of participants is eligible to submit a block. EC requires a secret, fair, and verifiable leader election. This is accomplished by the use of randomization in the election process. In the case of Filecoin’s EC, a DRAND beacon [19] is used to provide the seeds for an unbiasable randomization in the leader election process. The first step in the leader election process for a miner is to check if they are elected for the current epoch by running `GenerateElectionProof`.
Remember that a miner is elected in proportion to their quality adjusted power at `ElectionPowerTableLookback`. Setting `ElectionPowerTableLookback` requires that it be greater than finality. This is due to the fact that if `ElectionPowerTableLookback` is shorter, an evil miner might produce sybils with various verifiable random function (VRF) keys to enhance the likelihood of election and fork the chain to assign power to those keys.

This well-known attack in Proof of Stake systems would follow these steps:

1. Up until they discover a key that would give them the advantage, the miner creates the keys utilized in the VRF portion of the election.

2. The miner splits the chain and makes a new miner using the successful key.

In Proof of Stake systems, where the stake table is read from the past to ensure that no staker can move stake to a new key they find to be winning, this is typically an issue.As EC is a secret leader election, it is guaranteed that a winner is anonymous until they reveal themselves by submitting a proof, that they have been elected, a so-called `ElectionProof` using `GenerateElectionProof`. This outputs the

result whether a miner won the block or not as well as the quality of the block. The weight and block reward calculations employ an integer called `WinCount` where for example two blocks of quality "1" are equivalent to a `WinCount` of "2". This is done because a miner should not be able to divide their power among several identities and increase their chances of winning more blocks than if they kept their power under a single identity and particularly, one should not be able to implement tactic 2 below.

- Tactic 1: A miner with X% has the ability to run one election and take home one block
- Tactic 2: To win additional blocks, the miner divides its power to several sybil miners (with the total remaining at X%).

`WinCount` ensures that a lucky single block will receive the same reward as the miner would have received if they had divided their computing resources over several sybils.

A winning miner also needs to create a proof of storage, the so called `Winning PoSt`. The sector for which the miner needs to produce the proof for `Winning PoSt` is determined by randomness. A miner is not able to construct a block if they can not produce this proof within a set period of time. The sector is chosen from the list of sectors in the power table `WinningPoStSectorSetLookback` epochs in the past. Similarly to `ElectionPowerTableLookback`, `WinningPoStSectorSetLookback` must be set to a value grater than finality in order to prevent a miner from manipulation the power table and altering which sector is challenged (i.e., set the challenged sector to one of their preference).

One can verify if a leader `ElectionProof` is correct by the following checks

1. Check for the correct randomness with `GetRandomness(epoch)`
2. Verify the VRF correctness `Verify_VFR(vrf.Proof,beacon,public_key)` with vrf.Proof being the ElectionProof ticket
3. Verify the WinCount with `GetWinCount(vrf.Proof,miner,epoch)` with vrf.Proof being the Electionproof ticket

With these two proofs (`ElectionProof` and `Winning PoSt`) as a third step, a miner then can create a block with other transactions. In the same way that no miners may win in a round, there may also be several miners elected in the same round. As a result, many blocks may be built during a round. EC used all the valid blocks submitted within the time limits of a given round to form a tipset and every block within a tipset adds weight to its chain. The fork to be chosen to continue is the one with the highest weight, and if there are multiple Tipsets of equal weight, the one with the smallest final `ElectionProof` ticket is chosen. Even if by expectation at least one block will be created in every round, if no one generates one, one can just run the secret leader election of the next epoch with the new random seed.

All miners at round N will reject all blocks that fork off before round $N - F$ under EC's version of soft finality. Choosing such a F makes miner implementations simpler and ensures a macroeconomically-enforced finality with no additional cost to liveness in the chain, even if technically speaking EC is a probabilistically final protocol [11][21].

2.3 Technical basics

In this section we take a look at the technical basics of the software and technologies used for the implementation.

2.3.1 Go

Go or often referred as GoLang is a open source, compiled and statically typed programming language designed at Google, syntactically similar to C but with builtin memory safety, garbage collection and concurrency. The goal was to take the efficiency of a statically typed compiled language and combine it with the ease of programming of a dynamic language. While being efficient, providing fast compilation times, being type and memory safe, as well as having a good concurrency and communication support. Almost every modern software has a lot of dependencies, thus the build time depends heavily on managing these. What's why Go has explicit dependencies in source to allow for fast compilation and linking. As a solution for concurrency the concept of "goroutines" is introduced. Goroutines can be thought of as a lightweight thread managed by the Go runtime. The cost of creating such a Goroutine however is tiny compared to a thread and its common to have thousands of them concurrently. As said, Go has a runtime. The Go runtime is an extensive library that is part of every Go program, implementing all the critical features of the Go language, how every there is not something as a virtual machine such as provided by the Java runtime [28]. Go programs are compiled to the native machine code [18][9].

2.3.2 Lotus

Lotus is the open source [12] reference implementation for the Filecoin network, maintained directly by the Protocol Labs team. It is written in in Go (see Section 2.3.1). Lotus contains everything you need to run a Filecoin node or even start your own network. You can either just run a Lotus daemon to be a part of the Filecoin network, or you additionally append one or multiple Miners to that the daemon, to be able to generate blocks when having a winning ticket on the committed disk space. By the time of writing this, there are officially three other active implementation of filecoin, namely `Forest` a Rust-based implementation by a company called ChainSafe, `Fuhon` a C++-based implementation by a company called Soramitsu, as well as `Venus` another Go-based implementation maintained by the IPFS-Force Community. We focus on the usage of Lotus, as this is the reference implementation [21].

2.3.3 Redis

REmote **DI**rectory **S**erver (Redis) is an extremely fast open source [16] in-memory data structure storage often used as key-value database, cache or message broker. Redis also stores data on disk, but only to reconstruct the memory once the system starts [14]. Compared to a traditional relational database management system (RDBMS), Redis does not provide queries, but instead specific operations that are performed on the given data. Due to that, the information need to be stored in a way suitable for later fast retrieval without indexes, aggregations or other common features in traditional RDBMS. Developed by Salvatore Sanfilippo since 2009 out of the need for a better scalable database system for his startup, Redis became one of the most popular key-value databases. This grow comes mostly as a result of Redis's stability, power and flexibility in executing a wide range of data operations together with its famous speed. Speed not only by the fast execution times, but also speed in the sense that solutions with Redis can be built rapidly because of the ease in configuring, setting up, running and using Redis [30][15].

2.3.4 Containerization - Docker/K3s/K8s

Containerization is a specific form of virtualization, where not an entire system is emulated but instead only an isolated user space is put onto the same shared kernel. Everything an application needs to run from its binary to library, configuration or other dependencies is encapsulated and isolated within its container and with only limited access to the underlying resources. With that there is less overhead and less resources used and containers can be run on various types of infrastructure. From bare metal, in virtual machines or in a cloud environment. Also due to this high efficiency it is quite commonly used for packing up

many modern apps. Containerization evolved from `cgroups` a Linux kernel feature for isolating and controlling resource usage. From cgroups Linux containers (LXC) evolved with more features such as routing tables and the possibility to mount a file system. LXC is the basis for Docker, the most popular container technology and de facto industry standard for recent years even if the specifications are set by the Open Container Initiative (OCI). It is so popular, that it has become a synonym for containers, even if Docker is just one of the possibilities to run containers on one machine. To run a container an image is needed to act as a template on how to build a specific container. Thus someone booting a container can expect an identical experience no matter the underlying environment. When it comes to orchestrate containers at scale, Kubernetes (K8s) is the de facto standard because of its great flexibility and capacity to scale. K8s provides deployment patterns and takes care of scaling and failover. So if one container goes down, a new container needs to start and K8s manages that for you. Lightweight Kubernetes (K3s) is a K8s compatible variant packaged as one package, with minimized external dependencies and further enhancements to have a smaller footprint and better performance, with the goal to run on IoT, Edge devices and other resource constrained environments [10][29][3][2][22][26].

3 Design and Implementation

3.1 Filecoin

This section describes the relevant parts of the Filecoin specification to give some context to the following chapters. Unless otherwise noted, the information is taken from the Filecoin specification [21].

3.1.1 Storage Mining System

The Storage Mining System is a component of the Filecoin Protocol that is responsible for storing client data and producing proofs that demonstrate proper storage behavior by the storage provider. Storage Mining is one of the most critical component of the Filecoin protocol because all of the required consensus algorithms based on proven storage power in the network are based upon it. Miners are chosen based on the amount of storage power they have committed to the network to mine blocks and extend the blockchain. Storage is added in sectors (see Section 3.1.2), which are promises to the network that some storage will remain for a specified period of time. To participate in Storage Mining, storage miners must:

1. Add storage to the system, and
2. Demonstrate that they keep a copy of the data they agreed to keep throughout the sector's lifetime.

3.1.2 Sectors

On Filecoin, the fundamental storage units are called sectors. They come in predefined size and have commitments with clearly specified time intervals. The size of a sector balances usability and security issues. The storage market determines a sector's lifetime, which establishes the sector's promised lifespan. On the Mainnet sectors have a size of 32GiB and a maximal life time of 18 months.

When a sector is full (either with client data or as committed capacity), the unsealed sector is merged through the use of a Merkle tree into a single root (the so-called `UnsealedSectorCID`). An unsealed sector is subsequently converted into a sealed sector by the sealing procedure utilizing Concise Binary

Object Representation (CBOR). This conversion process is called "sealing" and a computationally demanding procedure that results in a unique sector encoding. It needs to be computationally demanding so that under no circumstances a miner can redo the procedure just in time for the Proof of Spacetime (see Section 3.1.4). Storage miners create a Proof-of-Replication (PoRep) (see fil:porep⁄) once data has been sealed, submitting the result using a Succinct Non-Interactive Argument of Knowledge (SNARK) to the blockchain as proof of the storage commitment, marking the sector as "ProveCommitted" from where on the respective WindowPoSt (see Section 3.1.4) becomes necessary.

3.1.3 Proof of Replication

A Proof of Replication (PoRep) is proof that a miner generated a unique replica of some underlying data correctly. In practice, the underlying data is the raw data contained in an unsealed sector, and a PoRep is a SNARK proof that the sealing process produced a sealed sector (see Section 3.1.2). It is critical to note that the replica should be unique not only to the miner, but also to the time when the miner created the replica, i.e., sealed the sector. This means that if the same miner creates a sealed sector twice from the same raw data, each time counts as a different replica. Miners must first produce a valid Proof of Replication before committing to storing data.

3.1.4 Proof of Spacetime

A Proof of Spacetime (PoSt) is a long-term guarantee of a miner's continuous storage of data from their sectors. This is not a single proof, but rather a collection of proofs submitted by the miner over time. A miner must periodically add to these proofs by submitting a WindowPoSt:

- A WindowPoSt is essentially a set of Merkle proofs over the underlying information in a miner's sectors.
- WindowPoSts compile evidence of numerous leaves from several sector groupings (called partitions).
- These proofs are submitted to the chain as a single SNARK.

Through historical and continuing submission of WindowPoSts it is guarantee that the miner has been storing and is still storing the sectors they agreed to store in the storage agreement. Once a miner successfully adds and "ProveCommits" for a sector, a sector is given a deadline, or a particular window of time during which PoSts must be submitted. A day is divided into 48 separate deadlines of 30 minutes and each ProveCommitted sectors gets assigned to one of them. Only the currently active Deadline may get PoSts. The time period for deadlines is 30 minutes, beginning at the "Open" epoch and ending at the "Close" epoch. Each PoSts is required to include randomness obtained from a random beacon (currently DRAND [19] is used). This randomness becomes accessible to the general public during the "Challenge" epoch of the deadline, which is 20 epochs before its "Open" epoch. Additionally, 70 epochs before its "Open" epoch, deadlines have a "FaultCutoff" epoch. Faults cannot be reported for the sectors of the deadline after this period. Before FaultCutoff deadline, a miner can declare a sector faulty with the incentive of getting a lower penalty fee than a with an undeclared fault.

3.1.5 Filecoin Actor

Actors can be seen as the Filecoin equivalent of smart contracts in the Ethereum Virtual Machine. As a result, Actors are critical components of the system. Any change to the current state of the Filecoin blockchain must be triggered by invoking an actor method. There are numerous different actors built into Filecoin, not all of which interact with the VM (see Section 3.1.6). Each actor has a so-called actor address which is generated by hashing its public key and a creation nonce. It should be consistent across all chain reorganizations.

3.1.6 Filecoin VM

The smart contract in the Ethereum Virtual Machine is analogous to an actor in the Filecoin Blockchain. The system component in charge of executing all actors' code is the Filecoin Virtual Machine (VM). On-chain executions, or the execution of actors on the Filecoin VM, cost gas. Any action taken (i.e., carried out) on the Filecoin VM results in a State Tree as the output. The Filecoin Blockchain's most recent State Tree is the current source of accuracy. A CID that is kept in the IPLD store serves as the State Tree's unique identifier.

3.1.7 Messages

A message is the fundamental source of state changes because it is the basis of communication between two agents. Message combines some tokens to be transferred from the sender to the recipient, as well as an optional or appropriate method to be used on the recipient. While processing a message that has been received, actor code may send subsequent messages to other actors. Since messages are processed synchronously, an actor must wait for a message to finish processing before taking over again. A message's processing requires compute and storage resources, both of which are measured in gas. The gas limit of a message sets an upper constraint on the computation needed to process it. The sender of a message pays for the gas units consumed by the execution of the message (including all nested messages) at the determined gas price. A miner selects which messages to include in a block from its Mpool (see Section 3.1.8) and is rewarded based on the gas price and consumption of each message, forming a market.

3.1.8 Mpool

The Filecoin protocol has a pool of messages called the Message Pool, generally known as Mpool or Mempool. It serves as the link between Filecoin nodes and the peer-to-peer network of additional nodes utilized for off-chain message delivery. Nodes keep a list of messages they intend to send to the Filecoin VM and add to the chain in the message pool. A message must first be in the message pool before it can be added to the blockchain. Even if the Mpool is "global", there isn't actually a central pool of messages that is kept someplace. The message pool is actually an abstraction that is realized as a log of messages that each node in the network maintains. As a result, a pubsub protocol is used to spread new messages throughout the rest of the network when a node adds them to the message pool. To receive messages, nodes must subscribe to the correct pubsub topic.

Since such a pubsub protocol takes some time to propagate messages, it takes some time for message pools at various nodes to synchronize. Though, it probably is never actually synchronized across all nodes in the network because of the constant streams of new messages being added to the message pool and the time it takes for messages to propagate. But as the message pool does not need to be synchronized, this is not an issue.

3.2 Test Network

This part describes the parts necessary to build a local Filecoin test network that allows gathering some measurements.

3.2.1 Test Case

We define a test case as to run the following procedure with specific parameters.

1. Run a Filecoin network with the predefined configuration (number of miners, their network topology, etc.)

2. Start applying a constant load of messages per second
3. Wait a predefined time (90 seconds) for buffers to fill and to reduce the impact of an "empty" network
4. Start with the measurement of all the variable we're interested in.
5. Wait for the specified experiment duration
6. Stop measurement and save the captured data and clean up for next run.

To get meaningful results each test case should be run many times in order to minimize the influence of random factors, which are necessarily present in a Filecoin, on average.

3.2.2 Lotus Customization

As described in Section 2.3.2 Lotus is the reference implementation for the Filecoin network. We use this implementation for our test network, but in order to get many details about the current state of the chain directly and quickly, we have modified Lotus slightly by allowing such data to be stored in a Redis [14] instance. In order to influence as little as possible with this data storage everything is executed asynchronously in go routines as shown in Listing 1.

```
go fil_benchmark.GetRedisHelper().RedisBlockFirstKnown(blk.Header.Cid().String(), blk.
    BlsMessages, blk.SecpkMessages, time.Now().UnixMicro())
```

Listing 1: Example go routine call to store when a block is first seen by a node

For our tests we gather the timestamp of each message, when it is entered into the Mpool (i. e. the moment at which the message is added to the pool of messages yet to be processed.), as well as the timestamp when the message is fully approved in a block on the chain. With that we can determine the delay of one message between sending a message and successful execution of the message. We define successful execution of a message as the moment it is applied onto the chains state machine. An other metric we gather is about the blocks, when is a block first announced in the network, and what message does it contain, as well as for each tipset, which blocks does it contain.

Another modification we made is not running the network with the default Mainnet parameter, as this would just be too resource intensive for the test hardware which is at our disposal. These parameters define some of the properties of the chain. For example there is a bunch of parameters specifying at which chain epoch a new Filecoin version should be applied, or parameters about the chain timing (what's the duration of an epoch, how long is the propagation delay, after how many epoch is a miner slashed, etc. But we don't just simply want to use the "2k debug" parameters provided for application developers, as we expect them to have a big influence on the Filecoin expected consensus algorithm by having an epoch time of just 6 seconds but still DRAND provides a new random beacon every 30 seconds. Thus we created our own set of parameters, originating from the Mainnet parameter. But for one, we can save all the updates done over time to the Mainnet and already start with the newest version at genesis time. However the main difference in the parameters is the sector size. While the Mainnet has a sector size of 32GiB (see Section 3.1.2), we reduce that to only 2KiB. This reduces required resources to run a Filecoin node dramatically and should give us the same results with regard to EC (see Section 2.2.1).

3.2.3 Containerization

We chose k3s as our containerization platform (see Section 2.3.4) to run our test network locally on one host (or in a cluster). The main reasons behind this decision was, the k8s compatibility allowing to change to a full k8s cluster with only minimal changes, the lightweight and speed advantages of k3s, since we

need to frequently create new networks. An other factor is, since running a Filecoin node is quite resource intensive, and running multiple nodes on the same hardware even more, the reduced resource requirement k3s are of course welcome there.

Our docker container image build process is split into 2 parts, the first part pulls the Github repository, checks out the defined branch or commit and builds the binaries. Since Go creates standalone binaries, we don't have to worry about linked libraries and we can just copy the newly built binaries into our image. The image is Ubuntu based, as the other Lotus requirements can be easily installed from the provided repository. We then further install our custom tools such as `rce` (see Section 3.2.4.2) or `rediscli` (see Section 3.2.4.3) and the custom container startup script initializing the Lotus node to the wanted Filecoin network.

3.2.4 fil-lotus-devnet

`Fil-lotus-devnet`[32] is basically just a series of scripts and program we have written to create a Filecoin test network for benchmarking. It contains scripts to build the custom lotus node container image, to launch a network but also to destroy a network. But it also contains some debugging tools as for example `node.sh` a script to get a shell on the specified node.

3.2.4.1 Lotus Node Startup Script

Each Lotus node container launches this script (`start_lotus.sh`) on startup. Node 0 has a special role, by that it generates a network genesis with the specified number of nodes already predefined as miners. We define all nodes as miner in genesis, to avoid having to do the lengthy and resource intensive process of sealing sectors and then committing them (see Section 3.1.2). Thus for each node keys are generated and then stored in a shared disk volume. We pre-seal 2 sectors for each node and all the relevant data is also stored on the shared volume, for the other nodes to access their information. This can just be done, since we are in an enclosed environment without any malicious actors. Once the genesis has been generated and shared with the other nodes, we can start the network. To ensure that we have one network and not every node is building and creating it's own local network, Node 0 is the first one to start and only after Node 0 has started, the other nodes continue. We intentionally do not use the automatic network connection via a so-called bootstrapping server, because the resulting network is always different and the measurements would be hardly reproducible. Thus depending on the defined network topology the nodes connect themselves to the desired partners. When the Lotus Daemon is successfully launched, we start the miner. Since we already defined them as genesis miner, the process of starting such a miner is much faster than creating a new miner from scratch. Once everything is running we launch the `rce` (see Section 3.2.4.2) to access the node remotly from out test coordination software (Section 3.3).

3.2.4.2 Custom Tool - RCE

RCE short for "remote code execution" is one of the small custom tools written in Go (see Section 2.3.1) and installed onto the custom lotus node and it does exactly what the name says. It provides remote code execution by listening for any http requests and executes the querystring as a command in bash. Thus if rce is running on `10.0.10.1` you could call

```
http://10.0.10.1?whoami
```

and the response would be `root` as rce needs to run as root to bind the ports and also to not have any permission problems when running any commands. `whoami` is not dangerous, but just call

```
http://10.0.10.1?wget%20http%3A%2F%2Fmalicious_source%20-O-%20%7C%20sh
```

which executes `wget http://malicious_source -O- | sh` as root and the machine is yours. This this should **never** be used in a non trusted environment, as anyone with access to that ip has unrestricted root access! You can run any command you want, the only requirement is, that it needs to be "url encoded" and as per specification the entire url must not exceed a length of 2048 characters. An example of an essential command used by `fil-benchmark` (see Section 3.3) is `lotus auth create-token --perm admin`

```
http://10.0.10.1?lotus%20auth%20create-token%20--perm%20admin
```

to get an API token which could then be used for the RPC API. It even is that sophisticated that if you have a longer running command, it returns a live stream of that commands output. Additionally to the remote code execution part, it also exposes the local Lotus RPC API to anyone. Meaning it basically opens a new port for everyone and not just the local interface and bidirectionally forwards the traffic between the newly opened port and the Lotus RPC API port.

The reason behind this tool is exactly what the name says, it allows for an easy remote control of the nodes from the test coordination software (Section 3.3). It allows for example direct access to generate a new API token to then connect directly to the Lotus RPC API.

3.2.4.3 Custom Tool - Rediscli

Rediscli is another super small custom written tool in Go (see Section 2.3.1) and installed onto the custom lotus node. It basically is just a small Redis client wrapper, allowing simple Redis usage from the shell or bash scripts. Its main purpose is to be able to use Redis as an easy method to communicate the start parameters to the new lotus nodes, as well as to coordinate the network startup sequence. It is based on `go-redis` a simple Go library facilitating the interaction with Redis within Go. Rediscli has some small very specific section as it looks whether an environment variable has been defined with where to find the Redis server, and alternatively assumes it is running on the host system and not in a container and thus tries to find out directly via K3s what the ip address of the Redis container is. The usage as a cli tool is really simple as it's a relatively intuitive command to read and write values (see Listing 2)

```
user@host:/path/to/dir[0] $ rediscli r myKey                # reading non existing keys
     gives exit code 1
user@host:/path/to/dir[1] $ rediscli w myKey "Hello World!" # writing a value to the
    specified key
user@host:/path/to/dir[0] $ rediscli r myKey                # reading from an existing
    key prints the value
Hello World!
user@host:/path/to/dir[0] $ rediscli d myKey                # deleting a key
user@host:/path/to/dir[0] $ rediscli r myKey
user@host:/path/to/dir[1] $ rediscli w character0 "Alice"
user@host:/path/to/dir[0] $ rediscli w character1 "Bob"
user@host:/path/to/dir[0] $ rediscli w spectator0 "Charlie"
user@host:/path/to/dir[0] $ rediscli w spectator1 "Dave"
user@host:/path/to/dir[0] $ rediscli s "character*"         # scan for matching keys
    and prints all of them
Alice
Bob
user@host:/path/to/dir[0] $
```

Listing 2: Example on how to read and write data from and to Redis with rediscli

3.3 fil-benchmark

`Fil-benchmark`[31] is the brain of our test bench. It is the software that connects all the individual parts together. It uses a `yaml` file with the specification of as an input, executes these tests and then outputs the results as a `csv` file.

The basic schema of fil-benchmark with the major interactions are illustrated in Figure 3.1. Note that the number of Lotus nodes depends on the number specified in the execution specification file.

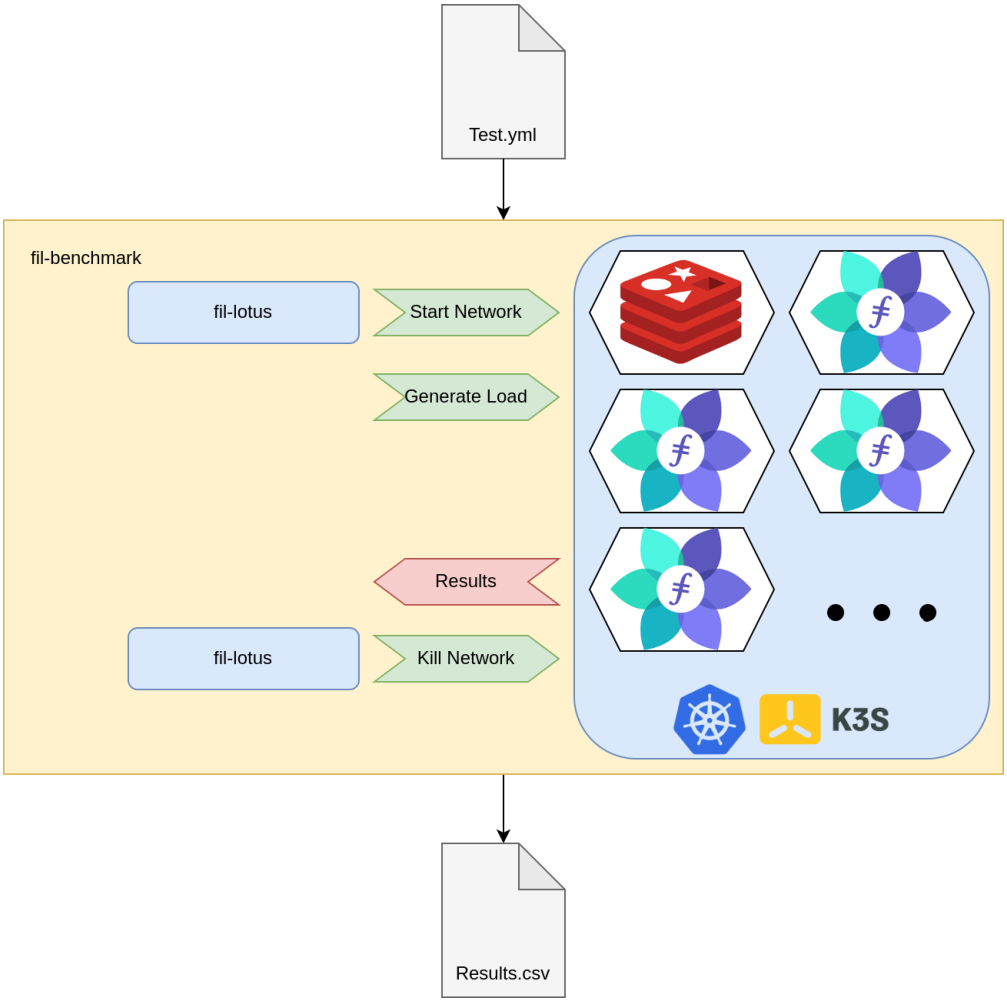

Figure 3.1: Schema and major interaction of fil-benchmark

Fil-benchmark iterates over all the test cases specified and runs them for the also specified repetition. One run starts with the start of the lotus test network. Once this network is up and running, the test procedure can start. For this the specified amount of load is put onto the network in the simplest possible form, by transferring funds between two wallets. To never limit ourself, the wallets and the fund transactions form a ring, i.e. One transaction transfers 1 FIL from wallet A to B, the next transaction transfers 1 FIL from wallet B to C, then 1 FIL from wallet C to D and so on, until we reach some wallet X where we transfer 1 FIL to wallet A.

In order to not influence the measured values with empty buffers, empty blocks and so on, we have a

90-second delay between starting with the load and start measuring values.

Once the specified time has passed, we stop the measurement, capture all the gathered data from our Redis instance and already make some statistics about this run, which are then written to a `csv` file. Afterwards we destroy the network before a new one is created in the next repetition or next test case.

3.4 Failed approaches

As part of the way to get to the testbed design mentioned in the previous sections, there were a lot of failed attempts. With this section we want to give an insight into what has lead to the final and successful design.

3.4.1 Testground

Testground [13] is a testing frame work for peer-to-peer systems initially designed to test IPFS. It seemed to have a lot of functionality but at the time, we were not able to get a Filecoin network running, thus we decided to switch to a custom system.

3.4.2 Add Miners On Runtime

It would be nice, if we had a predefined network we could just start and if a test case requires more miners, we just add them on the fly. Generally adding miners to a predefined network worked and we could add as many additional miners to the network as required by the test case (see Section 3.2.1), but the main issue was that it takes a lot of time and resources to add a new miner. In the process of adding a new miner, creating one is not the issue. The issue is with sealing a sector and committing that sector to the chain, which intentionally is a slow process not to repeat every few minutes. Removing sectors would be even more difficult, as you can not just remove a committed miner without consequences. Thus we needed to switch to adding all miners at genesis time, to shorten the process of starting additional miners, as they are already known to the network.

3.4.3 Reuse Network For Multiple Tests

It was always a concern for us, if it would be possible to reuse a network for multiple tests. The massive advantage would be that we would save a lot of time by not needing to recreate a new network when the network parameters do not change. To check if this would be possible, we ran throughput tests at a relatively mild rate for days and the results were more or less constant over the entire time. These results let us believe we could safely reuse a network and we did so for a long time without realising that after a running a network at its limit, we drag something along that massively influences the throughput.
To reuse a network we need to wait after one testrun until the network has processed all the load that has been input, because once a message has entered the global Mpool, it can't be removed. This limitations comes from the nonce, and with that the fact that messages from one origin need to be processed in order. The reason behind this limitation is to avoid the double spending problem. Another issue would be cleaning the global Mpool on all nodes simultaneously. With that we have a clean Mpool, but also a dead chain, as it also removes control messages and with that basically kills all miners. Thus the only option we have, is to keep track on what messages we have put into the system, what messages have already appeared on the chain and when all of the are processed we can continue with the next run. Even this waiting is faster than destroying and recreating an entire new network for each and every test case. But as said above, we somehow drag something randomly with us, creating totally non reproducible results. By randomly dragging something with us, we mean something from the previous runs influenced the results of the current run, with the effect of creating seemingly random results. Due to the previous test showing

us, reuse should not be an issue, this lead us down a deep rabbit hole of probably wrong hypothesis based on the influence of randomness in the entire system.

4 Results

As part of this thesis we wanted to use our testbed described in Chapter 3 to do some first analysis of the filecoin consensus algorithm in different situations. Always questioning why the results are like this. Does our testbed distort the results? Is it really the performance of the algorithm? Or are there other influences? These tests also helped to filter out the failed approches described in Section 3.4.

4.1 Testing Infrastructure

While we started testing on a powerful notebook, we soon realised, that more power is required to run a network with more than just 3 nodes. Thus the results of the testcases in the following Sections are all from running the testbed described in Chapter 3 on a VM on Amazon Web Service (AWS) either in the sizing of 'c5.4xlarge' or 'c5.9xlarge'. This means 16vCPU and 32GiB RAM resp. 36vCPU and 72GiB. The more powerful VM was used for all tests where testcases with 16 nodes are included (Section 4.2).

4.2 Testcase: Throughput With Different Network Size

Even if the algorithm is designed to work with hundreds or thousands of nodes, we want to know what influence the network size has on the throughput. Our tests have been running on a single large virtual machine, so there might be some differences over bigger networks when under heavy load. In Figure 4.1 are the average results of runs with the same parameters, on the same virtual machine with the only difference being the number of nodes in the Filecoin network. As expected the throughput grows linearly as long as the network has enough capacity to handle that load. Also the amount of time a message needs to be processed by the system is in a somewhat fluctuating but also low range. Keep in mind, a block is generated every $30s$, thus the expected average delay is around $30s +$ processing time. The more load, the more that delay grows. Interestingly the smaller networks can keep the delay on a lower level at higher load. This might come due to the fact that EC is expected to generate 5 blocks per epoch and thus it is likely that a majority of the 4 nodes is eligible to create a block and that this might have a positive influence on the delay of messages originating on the same node. Another reason might be the higher load on the VM with more nodes. Above an applied load of 30 messages per second, the amount of processed

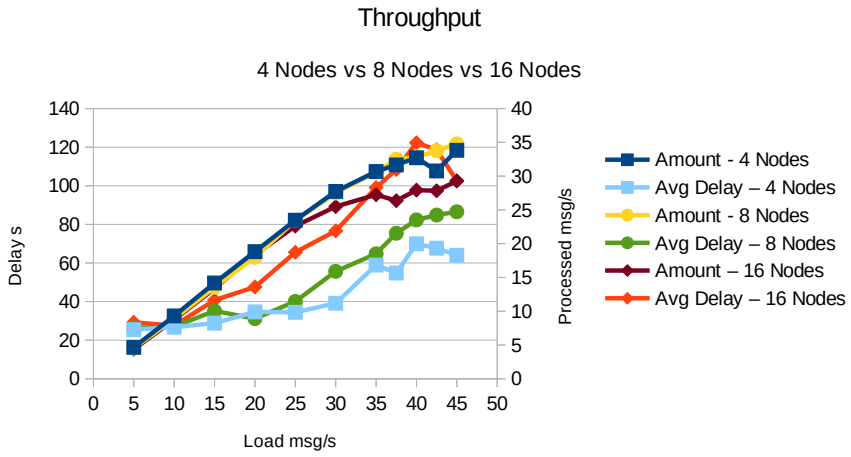

Figure 4.1: Chart comparing different network sizes

messages stagnates while the delay skyrockets (for this reason, we have not shown the values for 50msg/s and higher in the chart in Figure 4.1, as the whole graph would become unreadable). This until the point, where the underlying virtual machine doesn't has enough resources, event to that point where the only way to gain access again is to reboot, as no way of remote access was possible anymore.

4.3 Testcase: Throughput With Different Network Topologies

In Figure 4.2 are the average results of runs the same parameters, on the same virtual machine with their only difference being on how the 8 nodes are connected to each other. For one we have a star topology, where all nodes connect to node 0, while on the other hand we form a ring topology. This means node x is connected to node $x-1$ and $x+1$, as well as when we have n nodes, node n connects to node 0 to close the ring. All the connections are by design always bidirectional. The two compared network

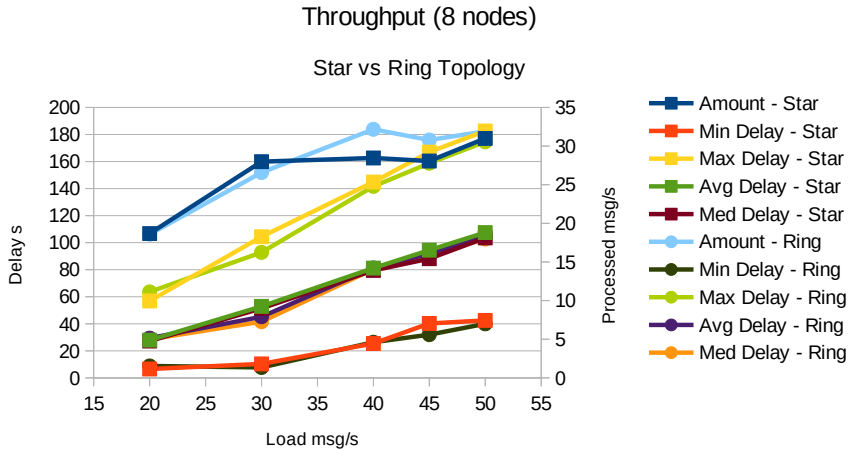

Figure 4.2: Chart comparing different network topologies

topologies perform very similar, with the main difference seeming to be the slightly higher number of process messages on high load in the ring topology. This might be caused by limited resources of the central node 0 in the star topology that needs to handle the traffic to all the other nodes, while in the ring topology we don't have such an overly important node in the network. In a ring topology, if one node is at capacity limit, we always have a second path for the information to flow. We are aware that the two topologies chosen do not exactly represent the typical peer to peer network topology. But they are two deterministic variants realizable on the used network size. In addition, the star topology in practice is starting to show up with large nodes that connect to an extraordinary number of smaller peers.

5 Conclusion

This thesis tackled the goal create a foundation to test the Filecoin consensus algorithm in a dedicated test network. fil-benchmark is a software built to allow tests of a local Filecoin network. Even if there were many challenges involved in testing such a complex system as the Filecoin blockchain network, we successfully implemented a testbed and got some results on the throughput.

5.1 Discussion

After many initial tries with inconsistent results, we managed to measure that on our test hardware a network of up to 16 nodes is capable of processing up to 30 messages per second consistently. If we put a higher load on the network we see some more messages might be processed, but not all of them. With that the average message processing time explodes as messages get queued.

We also found that a star topology performs worse at its limit compared to a ring topology. This is probably due to the resource limitations on the star's central node.

5.2 Future Work

As there now is a framework to allow Filecoin to be tested in a small test network, there numerous different parameters that could be changed and their impact evaluated. One could also compare different Filecoin versions against each other, to evaluate their performance difference in specific situations. Maybe one could increase this test network to more nodes, a hundred nodes or even a few hundred to thousands of nodes. How do the results change with such a change in size? Other possible future work might be analyzing a different part of the Filecoin system.

Bibliography

[1] Algorand. Algorant — the blockchain for futurefi. https://www.algorand.com, 2022. [Online; accessed 1-September-2022].

[2] K3s Project Authors. K3s: Lightweight kubernetes. https://k3s.io/, 2022. [Online; accessed 24-June-2022].

[3] The Kubernetes Authors. Kubernetes. https://kubernetes.io/, 2022. [Online; accessed 24-June-2022].

[4] Juan Benet. Filecoin: A cryptocurrency operated file storage network. *Filecoin website*, 2014.

[5] Juan Benet. Ipfs-content addressed, versioned, p2p file system. *IPFS*, 2014.

[6] Cardano. Cardano — home. https://cardano.org/, 2022. [Online; accessed 1-September-2022].

[7] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, 20(4):398–461, 2002.

[8] Peercoin Foundation. Peercoin - the pioneer of proof-of-stake. https://www.peercoin.net, 2022. [Online; accessed 1-September-2022].

[9] Google and many contributors. The go programming language. https://go.dev, 2022.

[10] Citrix Systems Inc. What is containerization? are there benefits of containerization? - citrix. https://www.citrix.com/solutions/app-delivery-and-security/what-is-containerization.html, 2022. [Online; accessed 24-June-2022].

[11] Nicola Greco et al Juan Benet. Filecoin: A decentralized storage network. *Filecoin website*, 2014.

[12] Protocol Labs and hundreds of contributors. Project lotus. https://github.com/filecoin-project/lotus, 2022.

[13] Protocol Labs and other contributors. testground. https://github.com/testground/testground, 2022.

[14] Redis Ltd. Redis. https://redis.io, 2022.

[15] Jeremy Nelson. *Mastering Redis*. Packt Publishing Ltd, 2016.

[16] Hundreds of Redis contributors. Redis. https://github.com/redis/redis, 2022.

[17] Meet Patel. Consensus algorithms in blockchain - geeksforgeeks. https://www.geeksforgeeks.org/consensus-algorithms-in-blockchain/, 2022. [Online; accessed 24-June-2022].

[18] Rob Pike. The go programming language. *Talk given at Google's Tech Talks*, 14, 2009.

[19] Ewa Syta, Philipp Jovanovic, Eleftherios Kokoris Kogias, Nicolas Gailly, Linus Gasser, Ismail Khoffi, Michael J Fischer, and Bryan Ford. Scalable bias-resistant distributed randomness. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 444–460. Ieee, 2017.

[20] Protocol Labs Team. Filecoin liftoff week, okt 19 - 23, 2020. https://liftoff.filecoin.io/, 2020. [Online; accessed 22-June-2022].

[21] Protocol Labs Team. Home — filecoin spec. https://spec.filecoin.io/, 2022. [Online; accessed 24-June-2022].

[22] Rancher Team. Rancher docs: K3s - lightweight kubernetes. https://rancher.com/docs/k3s/latest/en/, 2022. [Online; accessed 24-June-2022].

[23] Wikipedia contributors. Bitcoin — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Bitcoin&oldid=1077551584, 2022. [Online; accessed 17-March-2022].

[24] Wikipedia contributors. Blockchain — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Blockchain&oldid=1093369211, 2022. [Online; accessed 23-June-2022].

[25] Wikipedia contributors. Consensus (computer science) — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Consensus_(computer_science)&oldid=1091807218, 2022. [Online; accessed 24-June-2022].

[26] Wikipedia contributors. Docker (software) — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Docker_(software)&oldid=1094283134, 2022. [Online; accessed 24-June-2022].

[27] Wikipedia contributors. Interplanetary file system — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=InterPlanetary_File_System&oldid=1089110312, 2022. [Online; accessed 23-June-2022].

[28] Wikipedia contributors. Java virtual machine — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Java_virtual_machine&oldid=1094530700, 2022. [Online; accessed 24-June-2022].

[29] Wikipedia contributors. Kubernetes — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Kubernetes&oldid=1094587692, 2022. [Online; accessed 24-June-2022].

[30] Wikipedia contributors. Redis — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Redis&oldid=1083588417, 2022. [Online; accessed 23-June-2022].

[31] Marcel Würsten. fil-benchmark. https://github.com/fadnincx/fil-benchmark, 2022.

[32] Marcel Würsten. fil-lotus-devnet. https://github.com/fadnincx/fil-lotus-devnet, 2022.