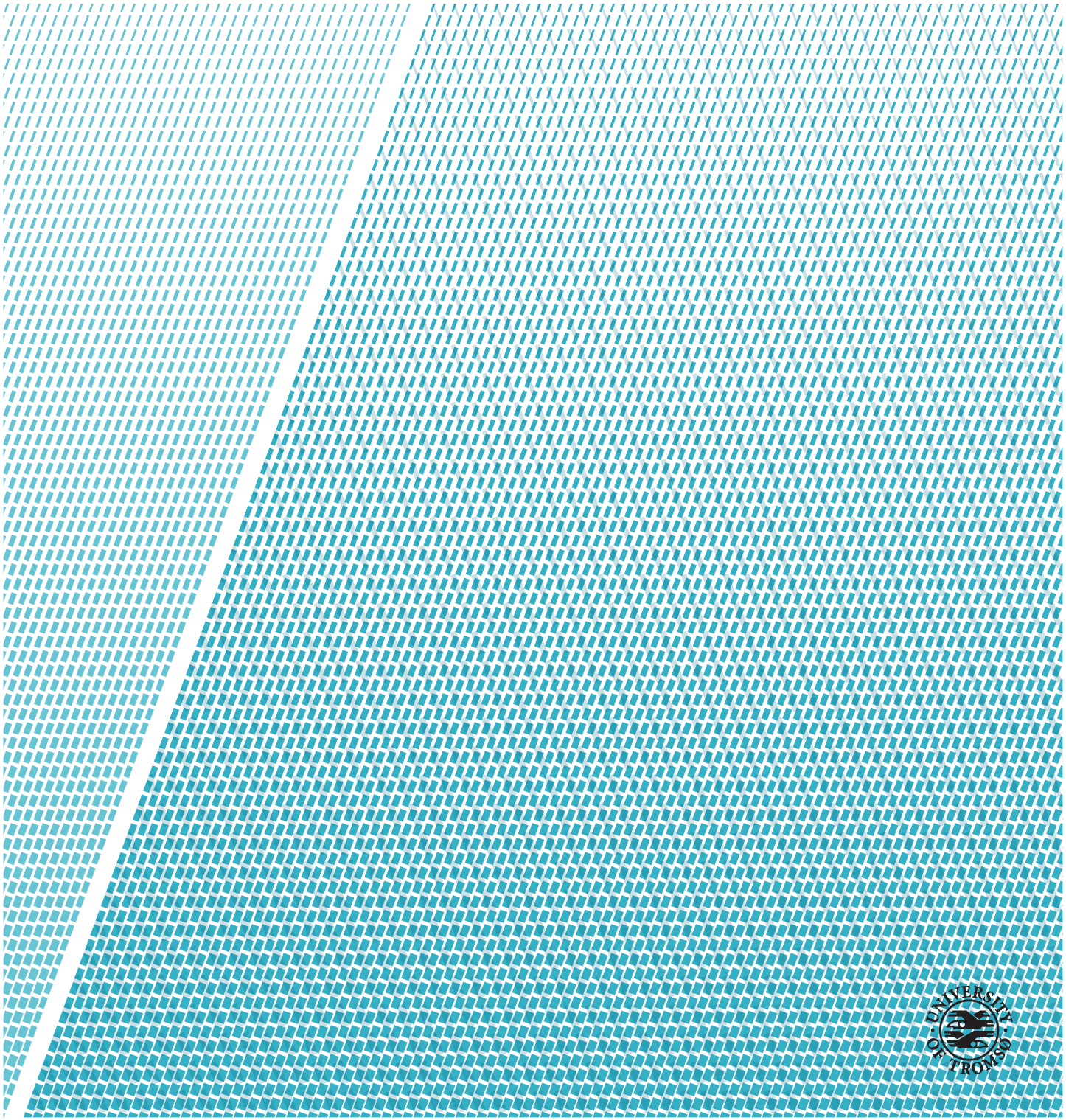
UiT
THE ARCTIC
UNIVERSITY
OF NORWAY

Faculty of Science and Technology
Department of Computer Science

FireChain: An Efficient Blockchain Protocol using Secure Gossip

Jon Foss Mikalsen

Master Thesis-3981 in computer science
June 1 2018



"There shall be no slides."
–Håvard D. Johansen

"Har du sett racken hannes?"
–Helge Hoff

"There are no mistakes, just happy little accidents"
–Bob Ross

"Det e bare å ssh'e sæ inn i HTML5 koden og memlocke stacken!"
–Eddie/Magga

"E du snart ferdig med mastern din i Outlook?"
–Eddie

"We're playing for all the marbles!"
–Unknown 4th grader

Abstract

Blockchains have become an integral part of many distributed applications, providing a new platform for interaction between system components. Blockchains are perhaps most known for their use in crypto-currency systems, such as Bitcoin and Ethereum, where pseudo-anonymous parties engage in transactions without a trusted third party. Blockchain systems often struggle to meet performance demands of real-word applications, rendering them inappropriate for performance sensitive applications. There is also concerns regarding the immense amount of electrical energy required to securely run existing public blockchain systems. Bitcoin alone consumes more than small countries. Private systems have higher throughput and avoid excessive energy consumption, but have closed membership and do not scale to the same extent.

Both public and private blockchains rely on some form of membership mechanism providing peers with a view of other participants. Existing systems often employ partial view protocols due to their natural scalability. However, recent work have shown that full view protocols are feasible in practice, and can scale to thousands of participants. With full membership, applications can send messages directly to their destination without any intermediate hops.

This thesis presents FireChain, which combines a Byzantine fault-tolerant gossip service and full membership, with a proposal for blockchain systems that does not consume excessive energy. We evaluate FireChain's performance through experiments on PlanetLab, and show that it scales beyond hundreds of members.

Acknowledgements

First and foremost i want to thank my supervisor Havard D. Johansen for your guidance, advice, and continuous feedback throughout writing this thesis. I also want to thank Dag Johansen for his contagious passion for computer science, which never ceases to amaze.

Further i want to thank my parents for proof-reading this thesis, and the members of the Corpore Sano research group for countless fun monday meetings, social meetups (beer), input, and feedback.

I want to thank my classmates, especially Christoffer Hansen, Kim Hartvedt Andreassen, and Helge Hoff for 5 years of shitz and giggles. And countless discussions about my thesis, this thesis would be pretty shit without you guys! I would further like to thank Helge for putting up with my shit for 5 years!

Lastly, i would like to give a BIG thanks to Erlend Graff for creating this fantastic Latex template, saving me countless hours fighting Latex.

Contents

List of Figures

List of Code Listings

List of Abbreviations

API Application Programming Interface

AWS Amazon Web Services

BA Byzantine Agreement

BFT Byzantine Fault-Tolerance

CA Certificate Authority

CPU Central Processing Unit

DAG Directed-Acyclic-Graph

DNS Domain Name System

DOS Denial of Service

DSA Digital Signature Algorithm

GB Gigabyte

GO Golang

GRPC Google Remote Procedure Call

GZIP GNU Zip

HTTP Hypertext Transfer Protocol

KB Kilobyte

KBPS kilobits Per Second

MB Megabyte

POS Proof-Of-Stake

POW Proof-Of-Work

RSA Rivest–Shamir–Adleman

RTT Round-Trip-Time

SGX Software Guard Extensions

TCP Transmission Control Protocol

TLS Transport Layer Security

UDP User Datagram Protocol

UiT University of Tromsø

VM Virtual Machine

1 Introduction

Blockchains are becoming an important building-block for distributed systems and Internet services. They provide applications with a distributed data structure, often referred to as a distributed ledger, where participants can interact without any form of trust [1, 2]. However, blockchains struggle with meeting throughput and latency demands of real world applications [3, 4, 5]. Existing blockchains, like Bitcoin [2] and Ethereum [6], are *permissionless*, or public: any process can join the system and execute protocol steps. Such systems are, however, susceptible to Sybil attacks [7] where the adversary controls a majority of nodes and thus dictates system state. To prevent such attacks, most permissionless blockchains deploy Proof-Of-Work (POW)-based consensus, often called *Nakamoto consensus*, where participants contribute computing power to further progress the chain. POW is the act of presenting proof that you have committed computing power to solve a cryptographic puzzle. Hence, an adversary's influence in the system is bounded by how much computing power he has, and not how many identities he is able to produce. POW chains provide fully open membership, but have throughput issues [2, 3, 8, 9, 4, 5]. With Bitcoin currently having a peak throughput of 7 transaction per second and latency of 60 minutes (if you follow the advised 6 block rule) [9].

Some blockchains are *permissioned*, or private: only selected peers are allowed to join the system, where identities of all participants are known. With admission control and known identities, private chains often employ classic Byzantine Fault-Tolerance (BFT) consensus [10, 11, 12, 13]. Permissioned blockchains such as Tendermint [14], Hyperledger Fabric [15], MultiChain [16], and Quorum [17]

employ BFT-based consensus, where participants co-operate in progressing the chain. They have significantly better throughput, but have a closed membership and scalability issues [3], typically not being able to scale beyond 100 members [10].

Most distributed systems, including blockchains, rely on some form of membership mechanism. Existing membership protocols such as Cyclon [18], Scamp [19], and Horus [20] all provide partial membership views. Systems such as Pastry [21] and Chord [22] incorporates a partial membership directly in their application. The main benefit of partial membership systems are increased scalability, both in terms of memory requirements and complexity of membership updates. They scale logarithmically in the number of participants. If a peer joins or leaves the network, not all participants need to be notified by the change in membership. Likewise, a single peer does not need to maintain information about all other participants, only a subset. By only maintaining a subset of the entire membership, peers do not need to receive all updates, thus, limiting bandwidth usage. However, partial views require all messages to be routed through the overlay network, increasing the chance of encountering a byzantine participant at each hop [23]. It was previously assumed that maintaining full membership views were infeasible due to bandwidth, memory, scalability, and handling frequent membership changes [24]. However, Fireflies [25, 26] and [23] have showed the feasibility of maintaining full membership views and their benefits, such as point-to-point messaging. Fireflies scales to thousands of participants, with memory requirement per participant around 600B, requiring 60 Megabyte (MB) for 100000 participants. Bandwidth usage, both under normal operation and when under attack, have been measured in [25] and deemed acceptable.

The Bitcoin specification [2] does not include a formal membership protocol. Participants form a Gnutella-like [27, 28] unstructured overlay network, which converges towards a full membership view. To join the network, peers contact a set of Domain Name System (DNS) servers which relays information of existing participants. Peers advertise addresses of already observed participants to neighboring peers, essentially flooding membership information. Hence, after joining, peers will continuously learn of other participants in the network. However, there are no measures to leave the network, peers might linger in the view of others long after they stopped participating. As observed by [29], at their time of writing 16000 peer addresses where being advertised, while only 3500 where reachable.

1.1 Problem definition

Proof-Of-Work (POW) and Byzantine Fault-Tolerance (BFT) chains are effectively on separate ends of a spectrum [3]. One scales, has open membership, but with low throughput, the other does not scale, has closed membership, but with high throughput.

We investigate if we can devise a scalable approach based on previous work, without the energy consumption of POW chains, but with configurable membership. Our thesis statement is that:

> *An efficient and scalable blockchain can be built using a Byzantine fault-tolerant gossip service and full membership.*

The protocol created and presented in [30] introduces another approach, basing a blockchain's consensus mechanism on *gossip* [31]. We aim to further build on this work and the earlier work of Fireflies [25, 26]. Fireflies is, as of our knowledge, the only Byzantine fault-tolerant membership protoco full membership protocol capable of scaling to a size similar to that of the Bitcoin network.

1.2 Scope & Limitations

This thesis does not intend to implement a fully functioning blockchain, our goal is to investigate the applicability of using gossip as a consensus mechanism. Evaluating all possible attack vectors and security issues concerning our implementation is beyond the scope of this thesis. We will introduce some possible attacks and discuss how they are handled. Also, we are not focusing on modifying the Fireflies protocol according to our specific use case. Part of our objective is to show Fireflies' applicability in a distributed application, hence, we do not want to tailor it specifically for our use. We assume that participants are not capable of breaking cryptographic primitives.

1.3 Methodology

In the final report by the Core of Computer science [32], the task force describes the discipline of computer science. From their findings, computer science is the systematic study of algorithmic processes, their theory, design, analysis, implementation and application [32]. With the fundamental underlying question: what can be efficiently automated?

Theory rooted in mathematics and consists of four steps; define objects to study, make hypothesis about the relations between objects, determine whether the predicted relations were correct, and interpreting results.

Abstraction rooted in the experimental scientific method consisting of four parts. Forming a hypothesis, create a model to test the hypothesis, design experiments and retrieve results, and finally analyzing result data.

Design rooted in engineering and follows four steps; requirements, specifications, design and implementation, and testing.

This thesis is rooted in *systems research*, which to some extent belongs to all three paradigms. We use existing knowledge of blockchains and Fireflies to devise a system with the intent of solving an existing issue in the blockchain design space. From this we design a prototype to develop and test. After developing a prototype, we design experiments to evaluate if our system solves the given problem.

1.4 Context

This thesis was implemented and written in the context of the Corpore Sano[1] research group. The Corpore Sano research group focuses on interdisciplinary research with computer, nutritional and sport science. With emphasis on personalized intervention technologies to improve health and wellness of people. We will now give a brief summary of previous scientific work done by the Corpore Sano research group.

Corpore Sano has done extensive international research of mobile agents and their applicability in the TACOMA project [33, 34, 35]. Mobile agents are processes capable of migrating to other hosts, typically in response to a client request. These agents are a powerful abstraction for distributed application developers, where the developer needs to know the location of some process. By attaching computations to agents, they can efficiently and seamlessly be moved across hosts and administrative domains. When deploying an agent at a site, to provide fault tolerance, an additional agent is deployed at another site. If the original agent fails, the backup can continue computation, or if its services is no longer needed, self-terminate.

As many Internet services regularly update content and their content is by nature dynamic and short-lived, the classical client-server model limits the

1. http://www.corporesano.no/

Internet's scalability. Corpore Sano developed the push based web-service WAIF [36, 37], which wraps Internet services in a push-based notification component. Instead of clients repeatedly pulling possibly new content from the server, the server notifies clients of new or modified content. Thus, relieving clients from constantly pulling for new updates.

After the blockchain technology emerged and quickly rose in popularity, Corpore Sano did a longitudinal study [4, 5] of its most popular application, namely Bitcoin. We investigated its scalability, performance, and cost. A notable discovery was that Bitcoin is becoming increasingly centralized, partially due to the emergence of mining pools. Instead of mining in a distributed fashion, as intended, miners now gather their computational power in pools, sharing all wealth from their block findings. This phenomenon emerged after Bitcoin became popular, resulting in an increase in computational power required to find blocks due to an influx of miners. Hence, the probability for single miners to find a block is not high enough to compensate for their committed computing power.

Corpore Sano has also presented work within the security and fault-tolerance domain. We created secure abstractions by embedding executable code fragments in protected capabilities, facilitating restricted data access across systems in cloud architectures [38]. Furthermore, we developed the intrusion tolerant network Fireflies [25, 26], which this thesis is built upon. Fireflies is a Byzantine full membership protocol capable of scaling to thousands of participants, we will go into further detail on Fireflies later on. With Fireflies, we created FirePatch [39], which disseminates time-critical software patches in a secure manner. By utilizing Fireflies' Byzantine fault tolerant full membership, patches can be disseminated to all honest participants in a Fireflies network through secure channels. Thereby, minimizing the attack window of an adversary aiming to delay dissemination of security patches.

We also developed StormCast [40], a distributed fault-tolerant weather forecasting system based on artificial intelligence. StormCast consists of co-operating agents, collecting, processing and exchanging weather data from fixed geographical locations. By coordinating weather agents, StormCast can forecast the weather at multiple geographical locations.

After the introduction of Software Guard Extensions (SGX) by Intel, Corpore Sano investigated the computational costs of using this new secure platform [41]. In SGX, threads execute within secure *enclaves*, shielded from the host's privileged software. However, shielding data and an execution environment from privileged software comes at a cost. We measured the architectural costs of entering, exiting, and copying data to and from the enclave.

Enforcing privacy policies after sharing data is non-trivial, and requires a substantial architectural design. We developed LoNet [42], a system which facilitates expressing data policies attached to files. Policies are programmable code, called meta-code, and are enforced by intercepting file system operations. LoNet also allows meta-code to affect derived data, for example, a coach could not be allowed to view a player's heartbeat data, but allowed to view health analysis data derived from the original heartbeat data.

Corpore Sano is currently working in close cooperation with the elite football club Tromsø IL, with existing systems such as Bagadus [43, 44, 45] and Muithu [46, 47] currently deployed. Bagadus [43, 44, 45] is a sport analysis system primarily focused on recording sport games and provide analytical feedback. Coaches can monitor real-time performance of their players and analyze team performance both at half-time and after matches. Muithu [46, 47] is a event based tagging system, providing coaches with ability to tag events as they occur on field. After tagging an event, typically by pressing a button on a pad, a recording of the preceding situation on-field is created and stored. Coaches can later view the recording, typically useful to further investigate certain situations. For example, tagging an event right after a player makes a vital mistake, which can later be brought up and analyzed. Within video analytics, Corpore Sano has developed a streaming system capable of keyword searching through videos [48]. Avoiding the need of download entire videos and searching through manually to find relevant content.

1.5 Outline

The rest of this thesis is organized as follows:

Chapter 2 introduces related work and background knowledge about related topics such as: Fireflies, blockchains, POW, Proof-Of-Stake (POS), and gossip protocols.

Chapter 3 Presents the system overview of FireChain.

Chapter 4 Covers the design and implementation of FireChain's communication substrate.

Chapter 5 Introduces FireChain's consensus component.

Chapter 6 Presents and discusses experimental results of our implementation.

Chapter 7 Concludes the thesis and discusses future work.

2

Background & Related Work

2.1 The Fireflies Membership Protocol

Fireflies is a membership protocol and gossip service developed at University of Tromsø (UiT) in collaboration with Cornell University. The first protocol version was published in 2006 [49], with several modifications proposed in 2008, named S-Fireflies [50]. However, these alterations were mostly rejected in 2015 due to practical concerns in a reworked version of the protocol [25].

The Fireflies protocol [25, 26] provides all correct member processes in a distributed system with up-to-date views on all other correct and stopped processes in that system. The protocol is resilient to Denial of Service (DOS) attacks and Byzantine faults, yet scales to support views with several thousands of members.

The protocol defines three data structures, an overlay mesh structure, and 10 rules for maintaining membership correctly. The overlay mesh structure consists of an arbitrary number of rings, each containing all members arranged in a pseudo random order. Nodes have exactly one successor and one predecessor in each ring. A node's successor and predecessor refers to their placement relative to the particular node in a given ring. Each node has a set of neighbors consisting of all their successors and predecessors in all rings. Nodes gossip with

their neighbors and are responsible for monitoring their immediate successors. By having all participants present in all rings, but in different orders, Fireflies enforces diverse neighbors.

If a node stops responding, its predecessor will accuse the node of being crashed by gossiping an accusation. Upon receiving a valid accusation, nodes start a local timer associated with the accused node, and at the time of expiration the accused will consider the node crashed. However, nodes can be falsely accused by being temporary unavailable or slow, therefore, nodes are able to issue a rebuttal if they receive an accusation concerning themselves.

If an attacker is capable of compromising membership information in an overlay network, he can effectively control it [25]. The goal of Fireflies is to prevent this by providing an intrusion tolerant network capable of operating in the presence of byzantine members.

2.1.1 Certificate authority

As with all distributed systems that relies on some form of agreement, Fireflies is vulnerable to Sybil attacks. Fireflies counter this by requiring members to obtain cryptographic certificates signed by a trusted Certificate Authority (CA). The CA is required to implement some means to limit the rate of Byzantine nodes joining the network.

All participants can be securely identified by their respective certificates. These certificates are distributed through gossip, hence, a node does not need to be in direct contact with a new participant to receive his certificate. A new participant will not be accepted by others unless they present a valid certificate sign by the CA. By requiring signed certificates, the identities of all participants are known and thus we avoid Sybil attacks [7]. The CA thwarts Sybil attacks by limiting an attackers ability to obtain multiple valid certificates.

2.2 Blockchain

Blockchains are distributed systems where member processes cooperate to maintain an agreed upon common append only data structure, often called a ledger [1, 2]. All participants keep a copy of the ledger, and execute a consensus protocol to validate and agree upon its content. Data entries are grouped into blocks, which in turn is built into a hash chain of blocks. Peers participate in the consensus protocol to agree upon the ordering of blocks and their contained entries. With the goal of progressing the chain, while preventing malicious

participants from altering already agreed upon content.

Blockchain systems can be split into two main groups; permissioned and permissonless. In permissioned systems identities of all participants are known and is not open for public access, there is a minor form of trust between participants. Permissionless systems are open for everyone, and identities of participants are not known. There are no trust assumption between peers in such systems.

2.2.1 Proof-Of-Work

As peers are anonymous in permissionless systems, they are susceptible to Sybil attacks [7]. Where an adversary attempts to gain control over a system by generating near infinite identities, thereby gaining influence and dictating system state. Hence, such systems need other mechanisms to fight these attacks.

Achieving consensus among a set of possibly malicious, anonymous entities is non-trivial. Byzantine Fault-Tolerance (BFT) consensus is susceptible to Sybil attacks [7] when participants are not identifiable. Proof-Of-Work (POW) consensus was introduced in [2], where participants solve cryptographic puzzles to further progress the chain. Participants present their verifiable solution to the cryptographic puzzle to other peers, proving that they committed computational power to solve it. POW acts as leader election, where the first peer to solve the current cryptographic puzzle, effectively finding the next block, has the authority to decide its content and is rewarded for his work. Thus, a peer's influence in the system is bounded by the computational power he is capable of generating, and not the amount identities he is able to create. To invalidate a committed transaction, an adversary would have to create a separate branch of blocks, often called a *fork*, and out-pace the main chain. Hence, after a transaction is *deep* in the chain, it would require a significant amount of computational power to remove it, ensuring that transactions are immutable and permanent after being committed. In Bitcoin [2], participants deem the longest chain, the main chain. After overtaking the main chain, honest peers would consider the created fork the new main chain, thus, invalidating blocks that were committed after the point of forking. Honest participants might also accidentally introduce forks by solving the current cryptographic puzzle simultaneously.

2.2.2 Proof-Of-Stake

The motivation for adopting a Proof-Of-Stake (POS)-consensus model is to avoid the significant energy costs of POW chains. In general, POW chains require a lot of electricity to maintain and progress. With the Bitcoin [2] network

currently consuming the same amount of energy as Ireland [51]. Also, in the absence of POW computations, a system might be able to increase throughput and decrease latencies significantly. However, implementing a faire and secure POS blockchain have shown to be complex in practice [52].

POS is similar to POW in that they both periodically elect a leader responsible for determining the next block. However, in POS, leader election is weighted by each participant's *stake* in the system, where *stake* represents how much resources a participant has invested in the system. Either electing a single leader for each time slot or forming committees of *stakeholders*, where *stakeholders* is the set of participants with the most stake. Stakeholders have a higher probability of being elected leader than participants with lower amount of stake in the system. In the context of crypto-currencies, stake could be determined by how much crypto-currency a peer is in possession of. Instead of participants competing for the authority to determine the next block as in POW, a peer is selected at random at every time slot, with peers weighted based on their stake in the system. When elected leader, a peer determines the next block, linking it to a previous one. If a malicious peer is elected leader, he could generate two blocks, introducing one to the full system and the second to a set of isolated peers. Hence, POS systems need a mechanism preventing malicious leaders from breaking the system.

Distribution of wealth is also an issue, if a single peer is in possession of nearly all the wealth, he will have a significant influence in the system. Although, peers with significant wealth are incentivized to operate honestly, if they are caught in being dishonest they are effectively undermining their own wealth. Blockchains based on POS include Blackcoin [53], Ppcoin [54], Ouroboros [55], and Algorand [8]. Ethererum [6] is currently based on POW, but there are proposals to adopt a POS and POW hybrid approach [56].

2.3 Bitcoin

The most common usage of blockchains is crypto-currencies, where Bitcoin [2] and Ethereum [6] are the most popular systems in use today. Crypto-currencies use blockchain to create a transfer log of some digital currency to and from accounts. Effectively creating a log of transactions between pseudo-anonymous entities, without any form of trust. They are unique in that they are the first technology to solve the double spending attack without any form of trusted third party.

Bitcoin [2] is based on POW-consenus, which is proven to converge except for a negligible probability [57]. Peers transfer *bitcoin* by creating and verify-

ing *transactions*. A transaction transfers bitcoin between one or more source accounts to one or more destination accounts [29]. An account consists of a public/private key-pair. To prevent forging of transactions, all transactions are signed by the sender's private key. Validation of transactions involve checking signatures and verifying that the sender actually has the amount of bitcoin he intends to transfer. Transactions essentially have a set of *inputs* which has to originate from previous transactions, and a set of *outputs*. These outputs can only be claimed once, and new ones are only created from new transactions. However, participants might receive transactions in different order. A transaction might claim an output of a previous one, which the participant have not yet received. Peers might also try to spend coins twice, referred to as a *double spending attack*. This could occur both by malicious intent or accident, where separate transactions try to spend the same output [29].

Transactions are disseminated in the network and included in everyones local ledger if valid. However, there needs to exist a commit mechanism to persist a set of transaction globally in all participants ledger's. Implementing a global commit mechanism in a distributed system with anonymous participants is non-trivial. Bitcoin periodically gather a set of transactions within a *block*, where each block links to the previous one by its hash value, effectively creating a hash chain of blocks. Blocks are introduced each time a participant solves the current cryptographic puzzle (POW), often referred to as *mining*. Participants that actively solve cryptographic puzzles are referred to as *miners*, and after solving a puzzle are paid for their efforts. The newly mined block and its content is then disseminated throughout the network, and included in everyone's ledger, effectively globally committing all the transactions present in the block. However, participants might solve the puzzle simultaneously, creating two valid system paths referred to as a *fork*. Bitcoin resolves forks by always following the longest chain, hence, participants are guaranteed that the main chain has the majority of computing power in the system. There have been proposals to change the chain selection algorithm, Ghost [58] suggests not only evaluating chains by their length, but the weight of their entire subtree. Thereby, an attacker cannot secretly create a hidden chain over time and introduce it later, effectively replacing a huge part of the previous main chain. This approach was later adopted by Ethereum [6].

Bitcoin disseminates membership, transactions, and blocks in a flooding-like fashion [29]. Peers advertise observed participants to neighboring peers, but there are no mechanism to leave the network or remove crashed peers. Hence, advertised peers are not necessarily still participating or alive. After verifying new transactions or blocks, peers advertise their availability to neighbors by issuing *inv* messages containing their ids. Subsequently, neighbors can request them by issuing a *getdata* message containing the ids of the ones missing from the their local storage. Thereby, blocks and transactions are constantly flooded

throughout the network upon creation. When two or more participants create a fork as explained earlier, a race begins to disseminate your block quickest. The more participants you spread your block to, the more peers will start working on your branch, hence, accumulating more computing power to your cause. With more computing power dedicated to your branch, the higher the probability that it will become the main chain. All blocks that are not building on the main chain are deemed invalid, likewise for their containing transactions.

2.4 Gossip protocols

The core concept of gossip protocols involve periodic information exchange between participants about recent events [31]. Participants periodically select a random peer in the system and reconcile information. By randomizing peer selection, as opposed to a fixed set, peers will eventually gossip with everyone in their view. With random peer selection and periodic reconciliation of information, all events will eventually spread to all participants with high probability.

Dissemination of data using gossip protocols have similar mathematically properties as how infectious diseases spread in a population. An approximation formula to estimate the fraction of infected hosts Y_r after r rounds, where each infected hosts infects f other hosts each round is given by: [31, 59].

$$Y_r \approx \frac{1}{1 + ne^{-f}r} \tag{2.1}$$

Where n is the total amount of hosts. The amount of infected hosts relative to uninfected ones increases exponentially each round by a factor of e^f [31]. In the context of gossip protocols, the formula describes how many hosts have received a gossip message after r rounds. A key detail from this equation is that the convergence of gossip among a set of hosts is correlated with the amount of gossip interactions each host initiate per round, and round frequency. Gossip protocols typically reconcile with only one host per round, thus, round frequency is in most cases the main convergence factor.

Gossip protocols have several strengths: they converge with high probability; impose a bounded load amidst frequent updates; not dependent on stable underlying network topologies [60]. Other classic distributed protocols, non-gossip based, can impose high workloads during frequent updates. By imposing a bounded load on participants, they will simply fall behind amidst frequent updates and converge when traffic decreases. These boundaries include bounded message sizes and gossip rates, how often participants gossip, and how much gossip each message can accommodate. Also, gossip protocols are able to oper-

ate on most underlying network topology, only requiring sufficient connectivity and bandwidth [60].

Gossip protocols also have limitations. Bounded message sizes and slow gossip rates limit the capacity of update propagation [60]. If the gossip content exceeds the maximum message size, it has to be split up into several messages and distributed over multiple gossip rounds, resulting in slower update propagation. This presents a trade-off between lower convergence times and increased overhead. By increasing the gossip frequency, one could facilitate frequent updates, but this would introduce more overhead. In extreme cases where the gossip rate approaches the network's Round-Trip-Time (RTT), resource contention at network interfaces might affect the protocol significantly [60]. Other weaknesses include not being resilient against malicious participants or faulty components executing the protocol incorrectly. If there is no method for verifying gossip data, malicious participants can pollute the system with corrupt data which becomes indistinguishable from correct data. Adversaries could attack a naive gossip approach by attempting to control information flow. By targeting the overlay network connecting peers, an adversary can separate participants into multiple partitions. After separation, the adversary can feed participants whatever information he sees fit, effectively controlling the network.

Gossip protocols are typically split into three types: dissemination (rumor-mongering), anti-entropy, and aggregate [60]. In dissemination protocols, peers only gossip about recent events, hence, propagation latency is an issue. If an event is not disseminated within a time frame Δ, where Δ is the amount of time a peer includes an event in its local gossip set, the event is lost. This could occur, for instance, due to network partitions, network outage, slow participants. Although, Δ would be reset for each time the event spreads to a new participants, given that Δ is started upon receiving the event, and not based on a timestamp contained in the event for when it occurred. Events also have high latency from when they occur to when they are delivered to all participants. Since peers only gossip about recent events, messages are small, thus, reducing bandwidth usage compared to anti-entropy.

Anti-entropy protocols reconcile state in each gossip interaction, typically used for data replication [60]. After peer a and b engages in a gossip interaction, both will have the same state. Messages are typically significantly larger than that of dissemination protocols since peers reconcile their entire state in each interaction, resulting in increased bandwidth usage. However, anti-entropy provides stronger convergence guarantees than dissemination since the entire state is reconciled in each interaction, and not just recent events.

Aggregate protocols aim to combine information from all participants and

compute some system wide value [60]. For example, like computing sum, max, median across the entire network. With highly scalable systems, computing aggregate values on data across entire systems are often more interesting than data at individual nodes [61, 62, 63].

2.5 Gossip in blockchains

Most existing blockchain systems use some form of gossiping to disseminate transactions, blocks, and membership information. By utilizing gossip, participants will eventually receive all transactions and blocks with high probability. For example, participants in Bitcoin gossip with neighboring peers about recent transactions, blocks, and advertise membership of other participants. Hyperledger Fabric [15] is a platform for deploying and operating permissioned blockchains. In Fabric, participants gossip about blocks, transactions, and membership information. Fabric divides gossiping into two modes: pull and push, where participants request state from other peers during pulling, and sends their state while pushing. Algorand [8] uses a similar gossip approach as Bitcoin, where participants select a small subset of peers to gossip with. However, in these systems gossip is typically used for disseminating blocks, transaction, and possibly membership information, and not as a consensus mechanism which we intend to.

The protocol presented in [30], is as of our knowledge, the only blockchain consensus protocol fully based on gossip. We base our consensus solution on the work presented in [30], where participants agree upon blocks by relying gossip that converges with high probability. We will be explaining our full design and implementation based on this protocol later on

2.6 PoW chains

Bitcoin-NG [64] is a scalable blockchain protocol with the same trust model as Bitcoin. They divide their protocol into two parts: leader election and transaction serialization. In Bitcoin, the miner who first solves the current POW effectively becomes the leader and serializes transaction history of the next block. Bitcoin-NG divides time into epochs, where in each epoch a leader is elected by solving a POW, as in Bitcoin. Blocks created from POW are called *keyblocks*, which does not include any transactions. Without content, keyblocks can be disseminated more efficiently due their reduced size. The leader then generates a series of *microblocks* containing transactions, these do not contain any POW and can thus be produced faster than *keyblocks*. Hence, transaction

latencies are bounded by network propagation delay of *microblocks* and the more infrequent leader election of *keyblocks*.

Ethereum [6] is a generalized blockchain platform for executing *smart contracts*, which essentially are distributed applications. Ethereum follows an *order-execute* paradigm where all smart contracts are ordered at all peers before executing them sequentially. After peers complete a POW, they sequentially execute all transaction within that block, likewise for all other participants receiving a new block. Thereby, all smart contracts are executed by all participants such that they converge to the same state. However, smart contracts are written by potentially untrusted developers. Hence, adversaries could execute a DOS attack by deploying a smart contract with an endless loop. To solve this, Ethereum introduces *gas*, which is payment for execution. The payment currency is Ethereum's native crypto-currency, *ether*. Thereby, the transaction issuer pays ether for having participants execute his smart contracts. All smart contracts are written in Ethereum's own scripting language Solidity, executed in their own Virtual Machine (VM) [65, 66]. By providing their own VM, Ethereum can determine execution costs and evaluate if smart contracts are deterministic. Ethereum [6] is currently based on POW consensus, but there are proposals to adopt a POW & POS hybrid [56].

Recent proposals, like Ghost [58, 67], Spectre [68], and Meshcash [69] aims to increase Bitcoins throughput by replacing the underlying chain structure with a Directed-Acyclic-Graph (DAG). With a DAG, chain selection algorithms can evaluate more metrics than just length. More specifically, Ghost [58] proposes to evaluate chains not only by length, but by the weight of their subtrees. From this, they are able to improve throughput significantly. These systems are based on POW consensus, where chain progression is bound by computing power. As our system does not intend to use POW consensus, the benefits of a DAG are not obvious, but could be explored more in the future.

2.7 PoS chains

Algorand [8, 70] is a relatively new crypto-currency, with transaction latencies in the order of minutes and throughput 125 times that of Bitcoin. They achieve this through the use of a Byzantine Agreement (BA) protocol to reach consensus among participants on new sets of transactions. To scale the agreement among many participants, only a selected few take part in each decision. Peers compute a Verifiable Random Function [71] to check whether they were selected to participate in the next BA. Algorand implements POS by assigning a weight to each participant based on their wealth. A peer's probability of being selected to participate in the BA protocol is thus directly correlated to their wealth. By

having a committee, effectively a small subset of the network with the most wealth, Algorand can reach consensus about new blocks and transactions in about a minute. Additionally, they avoid forks, even in the case where parts of the committee is malicious.

Algorand's gossip protocol is inherently susceptible to Sybil attacks [7], this is clearly stated in [8]. The paper does not specifically state whether Algorand targets a permissioned or permissionless deployment. If Algorand does not target a fully permissionless environment, similar to Bitcoin [2] where there are no elements of trust, Fireflies [25] would be an ideal candidate to provide a Sybil resistant gossip network. However, Fireflies requires a trusted CA, thus, the environment has to have one trust component. With Fireflies as the underlying membership and gossip protocol, Algorand would be Sybil resistant.

Ouroboros [55] is a recent proposal for pure POS blockchains, which rigorously prove their security guarantees. The set of peers with the most wealth in the system, namely stakeholders, participate in a coin-flipping protocol to select a leader for the current epoch. Ouroboros assumes that an adversary cannot corrupt participants for a duration longer than an epoch (e.g a day). POS chains are susceptible to *grinding attacks* where an adversary attempt to manipulate the randomness in leader selection to his advantage. Such attacks have a severe impact on POS chain that base their entropy on chain content. Ouroboros's joint-coin-flipping protocol does not depend on chain content and prevents adversaries from manipulating it.

2.8 BFT chains

Hyperledger Fabric [15, 72, 73, 74] is a open source system for deploying permissioned blockchains. Fabric expands upon what Ethereum does with smart contracts in a perimissioned setting. However, Fabric is novel in that it supports writing distributed applications (smart contracts) in general purposes languages such as Golang (GO) and C/C++. Other smart contract blockchain systems such as Ethereum [6] follows a *order-execute* paradigm, where all smart contracts are firstly ordered and validated before they are sequentially executed by all participants. Fabric introduces a novel *execute-order-validate* paradigm, essentially executing smart contracts in parallel before ordering them, increasing throughput significantly. Previous systems often hardcode their consensus protocols, while Fabric supports modular consensus. By providing pluggable consensus, applications can deploy protocols suitable for their deployment environment. Fabric achieves a throughput around 3500 transactions per second with sub-second latencies, and scales to over 100 nodes.

Fabric's membership service is not explained in detail, but it seems that participants gossip about recent membership changes, both leaving and joining peers. Hence, either participants are assigned to monitor other specific peers, or they randomly discover them to be unavailable when they try to gossip with them. We argue that Fireflies would be a fitting membership service in Fabric's permissioned setting.

ByzCoin [75] adopts the same approach as Bitcoin-NG [64], decoupling leader election and transaction serialization. They form a committee responsible for serializing transactions, where each peer's voting power is dependent on his recent hashing power contribution. The committee acts as a sliding window of participants, where only the most recent contributors are included. Hence, only peers that have recently contributed computing power is allowed to take part in the consensus protocol. When a peer finds a new block, he receives a *consensus group share*, effectively granting him more influence in the system. As a result, POW acts as *proof-of-membership* within the consensus committee, regulating voting influence between peers. Although ByzCoin employs POW, its consensus scheme is based on BFT with POW only serving as *proof-of-membership* and chain progression. By forming a consensus committee based on recent activity, ByzCoin can execute BFT consensus with a subset of recently active peers. ByzCoin also utilizes a collective signing mechanism CoSi [76], enabling the leader in each epoch to gather signatures of other participants in the consensus group in a scalable manner.

3

The FireChain System

To strengthen our thesis statement in Section 1.1, we have designed and implemented a blockchain system, called FireChain. This chapter describes the architecure of FireChain: a robust blockchain protocol based on a Byzantine fault-tolerant gossip and membership service. Unlike most blockchain based systems, FireChain does not rely on BFT agreement or Proof-Of-Work (POW) for consensus. Instead, it uses Byzantine fault tolerant gossip that converges with high probability. FireChain uses a Sybil resistant full membership protocol to provide participants with a full view of members in the system. From our full view we can efficiently determine other peers' view of the system and detect forks as they form. FireChain is not bound by the computational cost of POW, and can potentially progress faster than traditional POW chains, avoiding the excessive energy cost associated with POW-based chains. Such a reduction in energy requirements per operation can potentially yield higher throughput compared to existing blockchains. Typical permissioned (closed) blockchains that are not based on POW do provide better throughput, but do not scale beyond a few hundred members. FireChain uses a relatively open membership service and is able to scale beyond hundreds of members.

3.1 System overview

FireChain is split into three main components: consensus, state, and communication substrate. FireChain's communication substrate acts as a discovery service and provides a full system view, allowing FireChain components to seamlessly communicate with other participating peers. The consensus component orchestrates the system by using the communication substrate to contact and achieve consensus with other participants. Subsequently, after agreeing upon a new state, its stored in our state component, responsible for storing blockchain state. The overall architecture is shown in Figure 3.1. We will now briefly introduce all components, and dive further into details of our communication and consensus components in Chapter 4 and Chapter 5 respectively.

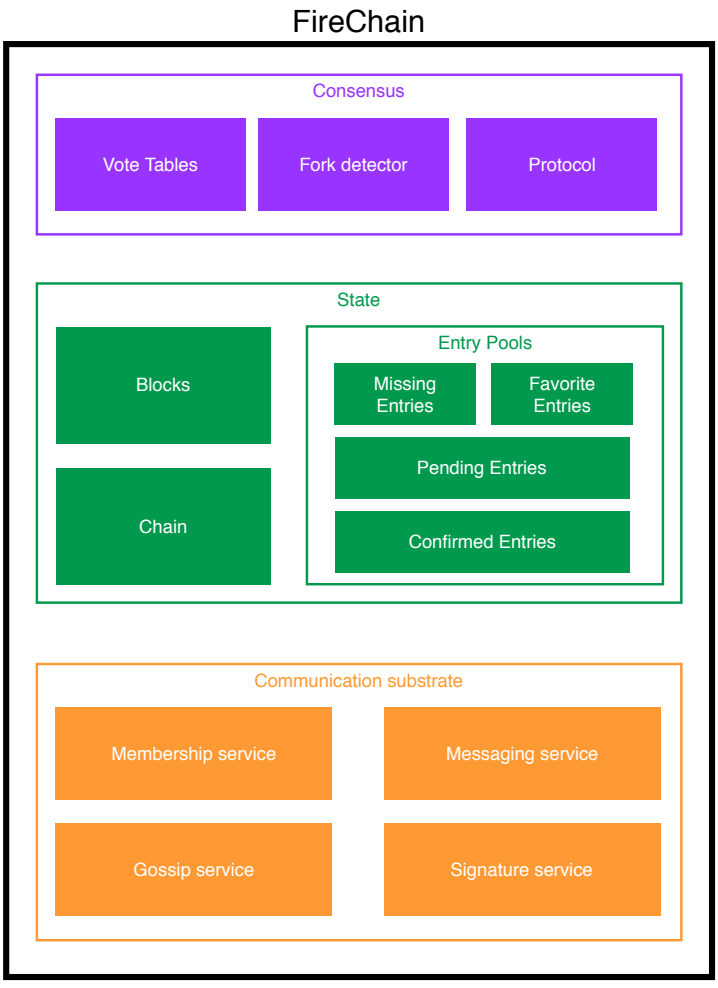


Figure 3.1: Architectural overview of FireChain.

3.2 State component

The FireChain state component keeps track of the entire blockchain state. This component is divided into three subcomponents: blocks, chain, and entry pool. The blocks component stores the set of all blocks, both committed and pending ones. The chain component contains the hash chain of committed blocks, effectively representing the blockchain state. Finally, the entry pool consists of a set of memory pools containing block entries.

3.2.1 Blocks

Blocks contain: a merkle tree containing all entries, hash of the previous block, and a block number, as shown in Figure 3.2. We use a third party merkle tree implementation.[1] Blocks have a fixed size limit, this was done for simplicity and dynamic sizes can be further explored in the future. As blocksize is decisive for block dissemination, our experiments will operate with different blocksizes to evaluate how the system behaves under different configurations. FireChain maintains an in-progress block, which is populated as entries are received.

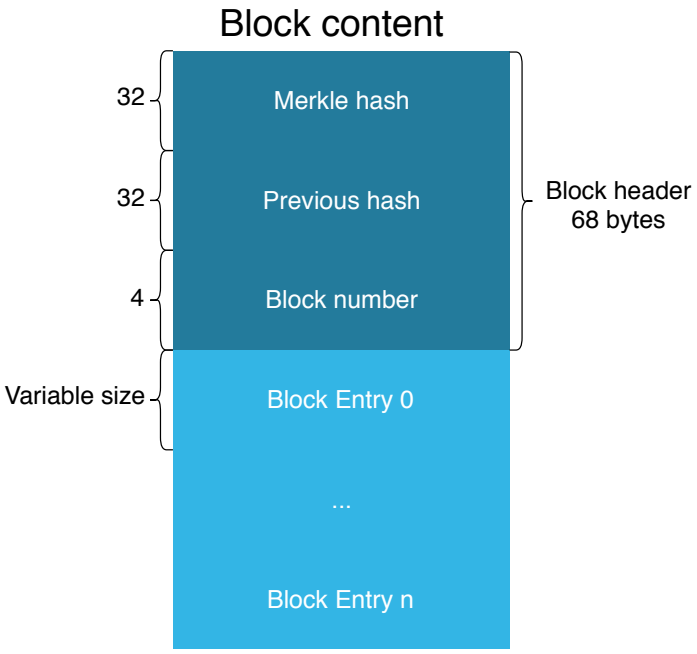


Figure 3.2: Content of a block.

1. https://github.com/cbergoon/merkletree

3.2.2 Chain

All committed blocks are organized in a chain of blocks, where each block links to the previous one, as shown in Figure 3.3, forming FireChain's *blockchain*. When resolving forks, we utilize the chain structure to determine which block the fork originated from. After finding the block, participants can reconcile their different branches and select the appropriate one as their main chain.

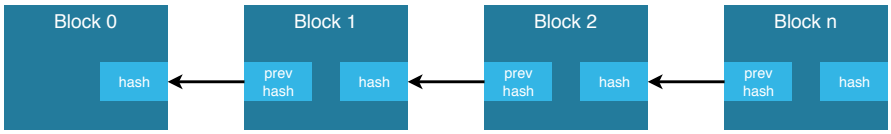


Figure 3.3: Blockchain structure. Genisis block (block 0) has no previous block.

3.2.3 Block entries

In our system block entries do not contain any vital data, we only want to investigate if our gossip approach is sufficient to ensure that all participants agree upon the ordering of blocks and their content. Hence, we do not evaluate the contents of a block entry, we only need an identifier for each entry. In crypto currencies, the id of a transaction is the hash of its content. We mimic the behavior for simplicity, each entry is a random sequence of bytes and the id is its hash. Entries are periodically created by each FireChain instances and added to their local pending pool. We deem it trivial to add data validation in future work as all other structures are in place.

In Bitcoin [2], the miner decides which transactions should be included in their newly found block and their ordering. Since we do not have miners, we need another mechanism to ensure that all agree upon the ordering of entries within a block. One could add entries in the order they are received. However, blocks with different entry ordering result in different merkle root hashes, hence, they are not considered equal. As a result, peers could populate their favorite block with the same entries, but consider other blocks with the same content as a completely different block. Therefore, we sort all entries in a block based on their hash values. If all peers sort their entries in the same manner, equal entries will result in an equal merkle root hash. Since our entries contain no vital data, and they have no relevance to each other this approach is feasible. When peers fill their local block, they first chose a set of random entries from the pending pool, then sort them. Thereby, we avoid favoring entries that have low hash values.

3.2.4 Entry pools

As multiple blocks might have overlapping entries, we want a mechanism to temporarily store them to avoid repeatedly requesting them. FireChain's consensus protocol defines a favorite block, consisting of entries that are currently being favored. These entries are more likely to be included in the next block, hence, if we simply discard all entries we are currently not favoring, we might need to request them later on due to changing favorite block. We effectively need a caching mechanism for block entries, trading memory storage for network usage.

Peers maintain several memory pools of block entries: pending, favorite, missing, and confirmed entries, as shown in Figure 3.4. The pending pool has a default size limit of 10 blocks and contains entries that have not yet been included in a committed block, similarly to Bitcoin [2]. The favorite pool contains the hashes of all entries that makeup the current favorite block, and the missing pool contains the hashes of the favorite entries that are currently missing from the pending pool. As peers change their favorite block, both the favorite and missing pool is reset and populated according to the content of the new block. Some entries might already be present in the pending pool, those lacking are added to the missing pool. Finally, the confirmed pool contains all entries that have been included in accepted blocks. Currently, FireChain does not persist blocks, they are only stored in the confirmed pool and in their respective blocks. Since our entries do not actually store any vital data, we deemed adding persistence a low priority. If FireChain is to be used, it is necessary to add support for persisting blocks.

Both the missing and favorite pool do not store entry content, only hashes. Hence, the missing pool reflects which entry hashes are currently in our favorite pool, but their content is lacking from the pending pool. In the scenario where a peer receives entries present in the missing pool and the pending pool is full, a random entry which is not present in the favorite pool is evicted from the pending pool. Thus, all entries present in the favorite pool have a priority in the pending pool. Since our favorite pool contains the hashes of entries in the current favorite block, the combined content size of its entries cannot exceed the blocksize. As a result, in a worst case scenario where a new favorite block is chosen with no overlapping entries with the pending pool, one blocksize of entries are evicted from the pending pool. With our memory pools, peers can identify which entries are currently in local storage, which ones are currently favored and those lacking to complete our favorite block.

Introducing memory pools can directly open for a DOS attack, where the adversary attempts to fill all participants pools with their own entries. However, since we prioritize our favorite block, an adversary cannot fill a participants

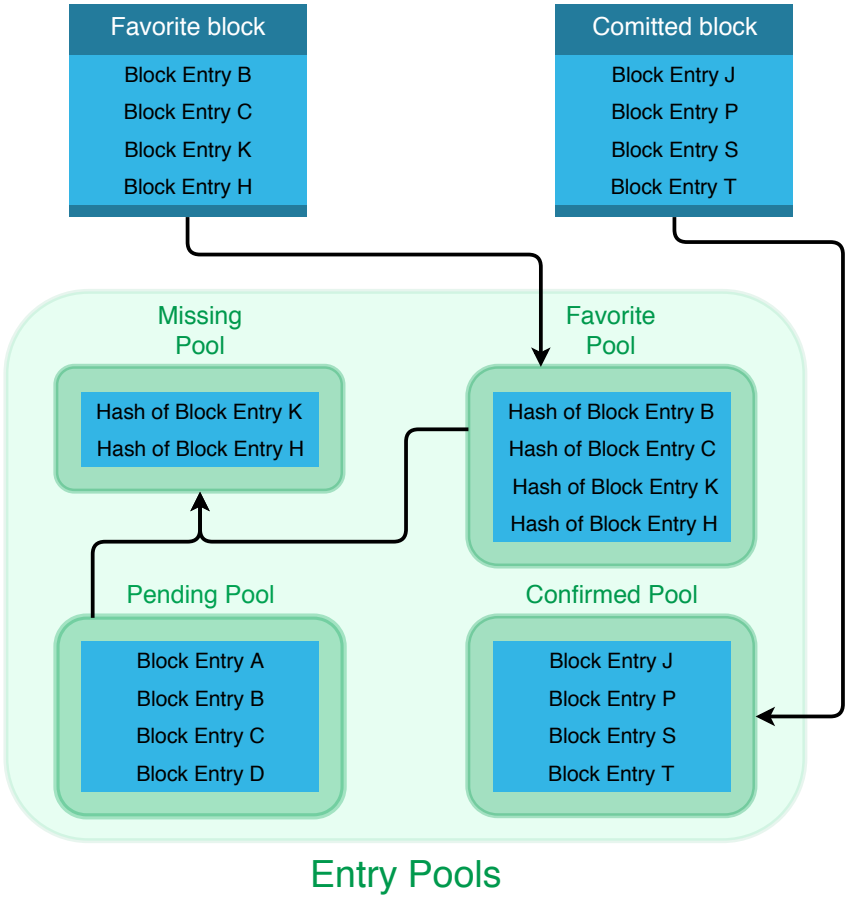


Figure 3.4: Structure of FireChain's entry pools.

pool with his own entries that are not currently favored by the participant. Peers will simply evict enough entries to make room for his favorite entries. Also, peers will only add entries in their memory pool that was at one point in time included in their favorite block. Hence, an adversary cannot simply generate entries and try to disseminate them amongst participants.

3.3 Consensus component

FireChain consensus is based on earlier work on gossip-based consensus [30]. Firechain's consensus component is responsible for reaching an agreement with other participants about the blockchain state. We reach consensus by periodically reconciling blockchain state with neighboring peers, eventually converging to the same state. Participants execute a consensus protocol, where

they vote and collectively agree upon the next state. After agreeing, peers commit the new state to the state component and repeat the consensus process. Membership and all communication between participants is orchestrated by our communication component. Hence, our consensus component is oblivious to how messages are transferred and how membership is maintained. The consensus component is the main orchestrator of FireChain, and uses our state component to track blockchain state and disseminates consensus information through our communication substrate.

Internally, the consensus component consists of three main subcomponents; a vote table, a fork detector, and a consensus protocol. The vote table keeps track of all other participants' votes, effectively representing all other peers' state. Our fork detector periodically checks whether we are progressing on a blockchain fork, and if so, resolves it. Finally, the consensus protocol forms the backbone of FireChain, and dictates how the entire system should operate. This component is further explained in Chapter 5.

3.4 Communication substrate

To provide Sybil resistant membership views, our communication substrate implements our own version of the Fireflies protocol [25, 26] called Ifrit (see Chapter 4). We form an overlay network where all participants are connected with high probability, and the diameter between two members is logarithmic in the number of participants. Subsequently, we use our implementation to provide a set of services; membership, gossip, signature, and a messaging service. The membership service provides a full view of other participating peers, and maintains system membership. The gossip service enables us to gossip with neighboring peers through secure channels. With the connected graph, the gossip service guarantees that all gossip will eventually spread to all participants. As gossip protocols are inherently susceptible to data corruption [60], our communication component provides a signature service, where all participating peers' signatures can be verified. Finally, the messaging service provides point-to-point messaging through secure channels. The Ifrit communication substrate is further explained in Chapter 4.

4

FireChain Communication Substrate

This chapter describes FireChain's communication substrate: a Byzantine fault-tolerant gossip and membership service, which we have named Ifrit. Ifrit is implemented as a library for the GO programming language. Ifrit improves on several deficiencies in the reference implementation of the Fireflies protocol.[1] First, he reference implementation was designed as a daemon process without a clear public API, making it hard to build applications on top of it. Second, it was implemented in Python with weak concurrency support, communicating with multiple peers concurrently is important for peer-to-peer systems. We therefore re-implemented it in GO, which is a statically typed language with builtin concurrency features. Ifrit provides an intrusion-tolerant overlay network were peers agree upon liveness of participants even in the presence of Byzantine members. The set of live peers form a connected graph where the diameter between two peers is logarithmic in the number of participants. Hence, all peers are connected and have a full membership view of the system, which eventually converges with high probability. Ifrit provides four services; membership, gossip, signature and a messaging service. The overall architecture is shown in Figure 4.1, with definitions of external interfaces. All internal interfaces combined implement the Fireflies protocol and maintain membership information. This is done regardless of application interactions through

1. https://github.com/joonnna/ifrit

the external interfaces.

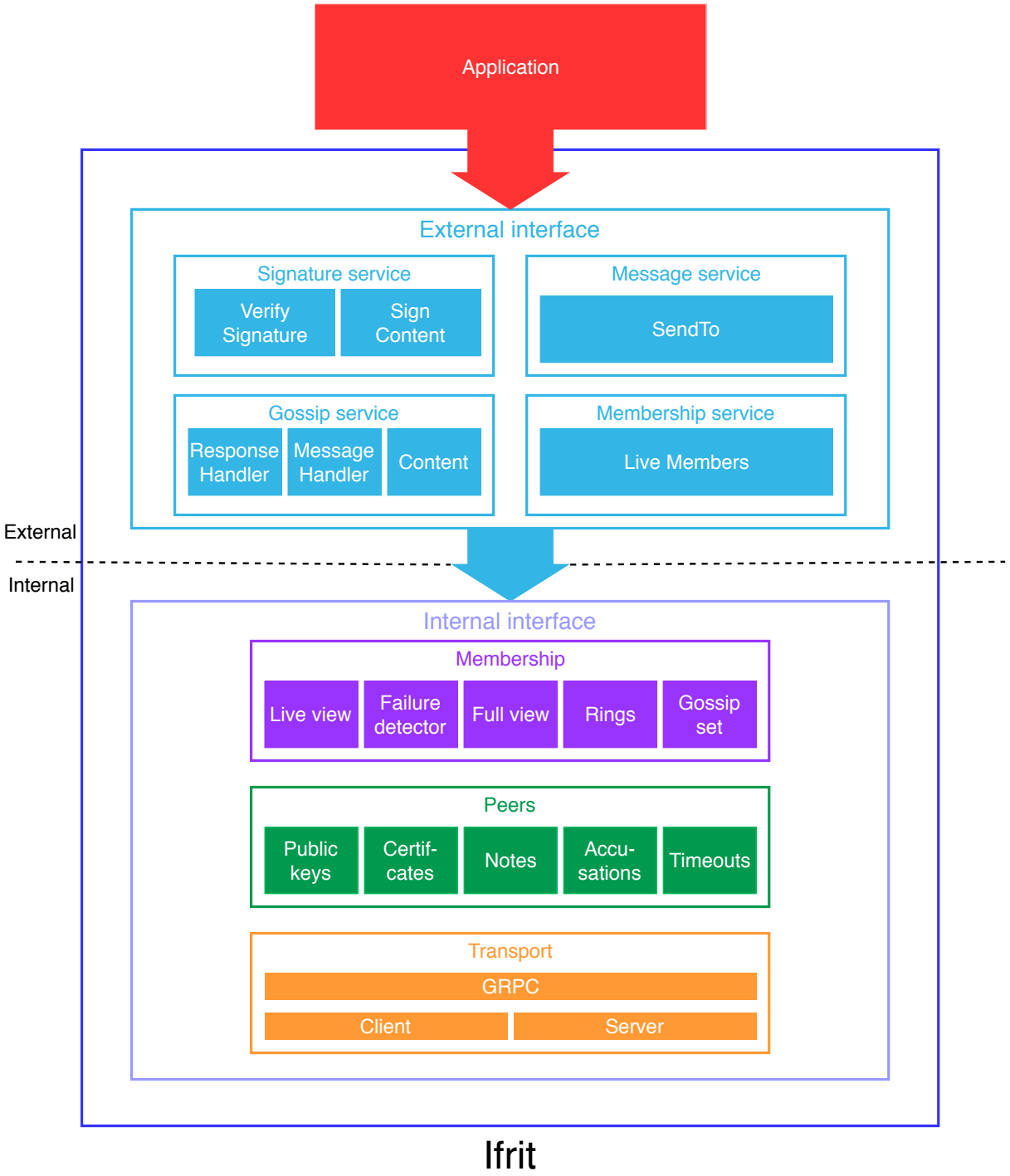


Figure 4.1: Architectural overview of Ifrit.

4.1 Data structures

Ifrit adopts the same data structures and rules as in the reference implementation by [25],[2] we will now explain them in further detail. Each peer maintains four data sets concerning other participants: certificates, notes, accusations, and timeouts. All gossiped data structures are signed and can be verified at the receiver, forming the first rule of Ifrit.

Rule 1 A *note* or an *accusation* is only valid if its correctly signed with the private key of its creator, and the creator's certificate is correctly signed by a common trusted CA.

4.1.1 Certificates

Certificates are X.509 compliant and used to establish secure gossip communication channels through Transport Layer Security (TLS). They contain node's unique identifier generated by the Certificate Authority (CA), their public key, and their network address, effectively binding their public key to their identifier and network address. Also, the amount of gossip rings are stored in a X.509 extension field. Certificates are gossiped between nodes, upon receiving one, its signature is verified to ensure validity. By having them signed by commonly trusted CA, we ensure that all of its content is tamper-proof and can securely identify participants. The purpose of certificates is to provide an immutable data structure that can uniquely identify nodes, and facilitate TLS communication.

4.1.2 Notes

A *note* represents a life signal from a node, containing: an epoch, a mask, a signature, and the node's unique identifier as shown in Figure 4.2. Epochs are monotonically increasing counters, establishing the order of notes received from a particular node. Only the most recent note is deemed valid as described in Rule 2. The mask is a bit field representing enabled rings. If a node is falsely accused by its predecessor, it can disable that specific ring by setting the corresponding mask bit to zero. Effectively informing other nodes to ignore accusations concerning itself originating from its predecessor on that ring. When falsely accused, notes are used to rebut the accusation by incrementing the epoch number, invalidating the accusation as described in Rule 5.

Rule 2 A *note* from node p is only valid if its the most recent observed note

2. https://sourceforge.net/projects/fireflies/

from p.

Rule 5 Upon receiving a valid *accusation* concerning itself, a correct member will immediately gossip a new note with an incremented epoch. Thereby invalidating the *accusation* at all other correct nodes.

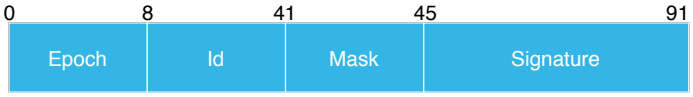


Figure 4.2: Note structure, numbers above fields refer to where they start.

4.1.3 Accusations

An *accusation* contains the epoch of the accused's most recent note, a ring number, a signature, and the accuser's identifier as shown in Figure 4.3. When a node stops responding to pings, its predecessor creates an accusation which will eventually be disseminated throughout the network. The ring number represents which ring the accusation originated from, this is necessary to determine validity according to Rule 8. An accusation is only valid if the note associated with it is valid, and the accuser is the direct predecessor of the accused as described in Rule 3 and Rule 8. Therefore, when a rebuttal is issued with a higher epoch number, the accusation becomes invalid due to the note associated with it becoming invalid.

Rule 3 An *accusation* is only valid if the associated note is valid.

Rule 8 An *accusation* is only valid in ring r if the accuser is the direct predecessor of the accused in ring r.

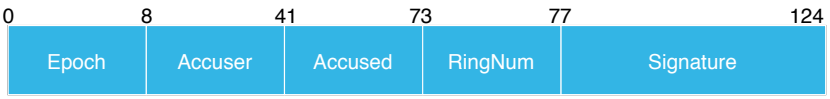


Figure 4.3: Accusation structure, numbers above fields refer to where they start.

Originally, accusations contained the most recent note of the accused. To reduce bandwidth requirement of gossiping accucations, this was changed to only contain the most recent note's epoch, identifier, and ring number. This reduces the size of an accusation with 56 bytes. This improvement does not reduce the systems security properties since tampering with note information such as the epoch number is not helpful in trying to falsely accuse peers. For instance, accusations with higher epoch numbers that the current note will simply be ignored by correct members. Such accusation also constitutes a proof

of malicious behavior, and can be used to expel the offending member. Since participants is already in possession of each other's note, when determining accusation validity, all note information will be retrieved from the accused's own signed note. If a node is not in possession of the accused peer's note, the accusation is discarded.

4.1.4 Timeouts

The timeout data structure is only kept as part of local state, and not disseminated to other members. They contain the last note from the accused node, the observer, and a timestamp from when the timer was started as shown in Figure 4.4. A timeout is only valid if its associated accusation is, as described in Rule 4. Therefore, if a rebuttal is received before the timeout expires, invalidating its accusation and subsequently deleting the timeout. Timeouts are started as a result of receiving a valid accusation, if the timer expires, the accused node is considered crashed and removed from the local node's live view. However, the crashed node is not removed from the full view, enabling him to rejoin the network without having to re-disseminate his certificate throughout the network.

Rule 4 A *timeout* concerning node p is only valid if there exists a valid *accusation* for p.

Figure 4.4: Timeout structure, the observer field is a local representation of a remote peer. There is no need to show byte indexes here since timeouts are not gossiped.

4.2 Gossiping

All data structures are gossiped (except timeouts) in the following order: certificates, notes, and then accusations. This way, nodes will always be able to recognize the unique identifier present in both notes and accusations. Notes and accusations are signed with the private key of their creator, such that nodes can verify their integrity by using the public key stored in the creator's certificate. Hence, all gossip messages are tamper-proof, and if tampered with, they are discarded as described Rule 1.

Ifrit imposes a set of rules concerning gossip partners:

Rule 9 For each ring *r*, correct members maintain a secure gossip channel to their neighbors (successor and predecessor) in ring *r* which are considered live.

Rule 10 For each ring *r*, correct members only accept gossip from their neighbors in each ring *r* which are considered live.

After converging to the same view, participants will only try to gossip with their correct neighbors. Thereby, discarding traffic originating from malicious or simply incorrect nodes. As a result, participants will disconnect gossip channels upon a change in neighbors, where the old neighbors either crashed or is temporarily unavailable. Upon resurfacing, a peer re-integrates into the network by proving his liveness, only then will participants accept gossip connections from him. To prove his liveness, the peer has to rebuttal the accusation that was created when he disconnected. When contacting his old neighbors, they should inform him of the accusation concerning him, such that he can rebuttal it, and thus re-integrate into the network.

4.2.1 Rings

A ring consists of all members in the Ifrit network organized in pseudo-random order, an example of a gossip mesh is shown in Figure 4.5. Each ring have a different ordering of members to enforce diverse gossip partners and monitors. Random gossip partners are essential for providing the high probability of convergence property of gossip protocols. All participants gossip with their closest neighbors in each ring, namely, their successor and predecessor. Also, participants monitor their successor on each ring. In Figure 4.5 member *A*'s gossip partners are highlighted in red, each one of them are either its predecessor or successor in their respective ring.

With different monitors in each ring, Ifrit guarantee, with high probability, that all participants have at least one correct monitor. A node *p* can disable rings with misbehaving predecessors by setting the appropriate bit in the mask bitmap in its own *note* to zero. By disabling a ring, all other nodes will ignore accusations concerning *p* originating from the disabled ring as presented in Rule 7.

Rule 7 An *accusation* is only valid in ring r if the bit corresponding to r's ring number in its contained note is enabled.

Hence, preventing corrupt nodes from generating additional traffic by repeatedly accusing their successors. However, a node might disable all of its rings by accident, or a corrupt one might disable all on purpose. Therefore, Ifrit

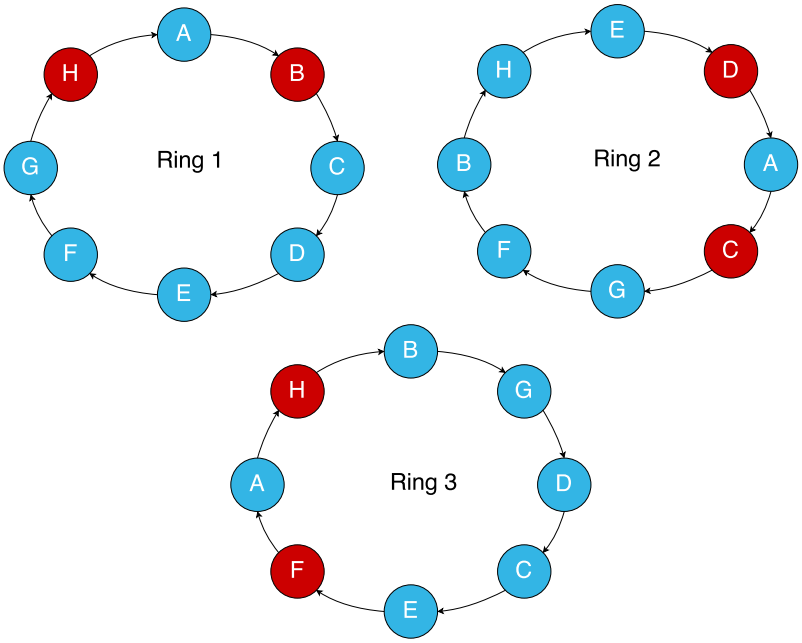


Figure 4.5: An example of a gossip mesh consisting of three rings. All members colored in red are neighbors of member A, forming its set of gossip partners.

impose an upper limit on how many rings can be disabled at any given time. The amount of rings k can be formulated as $k = 2t + 1$, where t is the upper bound on ring deactivation. From this, Ifrit presents Rule 6:

Rule 6 A *note* is only valid if the contained bitmask is of length 2t + 1 and at most t of the bits are disabled.

As a result, correct nodes cannot deactivate all their rings by accident and corrupt ones cannot deactivate all on purpose. For example, with 11 rings, participants can only deactivate 5 rings. If a node wants to disable ring a, but has reached his deactivation limit, he reactivates another ring b, and deactivates a. This is done in a round-robin fashion, not prioritizing any rings. However, this could be extended by keeping a small history log for each ring, where nodes could prioritize deactivating rings with a higher frequency of misbehaving monitors. Thus, punishing byzantine behavior.

4.3 The Membership service

Ifrit's membership service provides applications with the current live view of the system. We expose an endpoint where peers can retrieve the addresses of all participants currently believed to be alive. This is the primary purpose of Ifrit, providing a BFT full membership view. Participants that stop responding, either by temporarily becoming unavailable or crashing are removed from the live view. Hence, our live view only contains active participants. As of now, our membership endpoint retrieves all live members, hence, each time the application needs an updated view it has to fetch the entire membership. Ideally, we want to provide both the option of fetching the entire view and periodically receive updates concerning leaving and joining peers. We could implement incremental updates as a subscription based service, where applications subscribe to membership changes. Each time a peer joins or a live one crashes, the application could be notified with a message containing the peers address and live status.

Ifrit maintains three structures for membership management: a full view, a live view, and a set of ring structures. The full view contains all peers ever observed that possessed a valid certificate. The live view is a subset of the full view, containing only the peers believed to be alive. Finally, the ring structures are used for gossip and monitoring purposes. A peer's position within the rings determines his gossip partners and monitoring responsibilities. Rings contain all peers present in the live view, and the number of rings used is a configuration variable determined at startup. Hence, peers are removed and added to the rings depending on changes in the live view. Furthermore, rings facilitate accusation invalidation, when adding a node to a ring, gaining a predecessor p and successor s, if there exist an accusation concerning s issued by p it can simply be removed due to it becoming invalid since p is no longer the direct predecessor of s. The pseudo code can be seen in Code Listing 4.1.

Code Listing 4.1: Adding of live peers.

```
1  func addLivePeer(p *peer) {
2      // Rings contain all live participants.
3      for r := range rings {
4          // The peer gains a successor and predecessor on each ring,
5          r.add(p)
6          successor, predecessor := p.neighbors(r)
7
8          // If the new peer is placed between two participants
9          // where there already exists an accusation.
10         // We can now invalidate it, since the new peer
11         // is now the direct predecessor of the accused
12         if predecessor.accused(successor) {
13             successor.removeAccusations()
14         }
15     }
16 }
```

Upon receiving a new valid certificate, participants create a local peer representation and add it to their local full view. The new certificate is included in participants' gossip set, however, the peer is not considered live. A peer is not considered live before participants have both a valid certificate and note from the respective peer. Upon receiving a valid note, the peer is added to the live view and ring structures, hence, it its now considered live. When viewed as live, neighboring peers will open gossip connections and the peer will eventually learn of all other participants in the network. Since Ifrit provide a full membership view, and peers can deterministically determine who they should be gossiping with, a peer that is considered live by the network will always be contacted by his neighbors.

4.3.1 Joining the network

To join an Ifrit group, a peer *i* sends a certificate signing request to the CA. If accepted to join the group, the CA responds with a signed certificate, and a list of certificates of peers already participating in the network. *i* then contacts the closest neighboring peer in the list received from the CA, which in turn finds his actual neighbors. Finally, peer *i* can contact his appropriate neighbors and integrate into the network. Our CA acts as an entry point into the network, supplying new peers with certificates of existing participants. Peers discover other participants through gossiping with the peers belonging to the supplied certificates. The process of joining the network is depicted in Figure 4.6

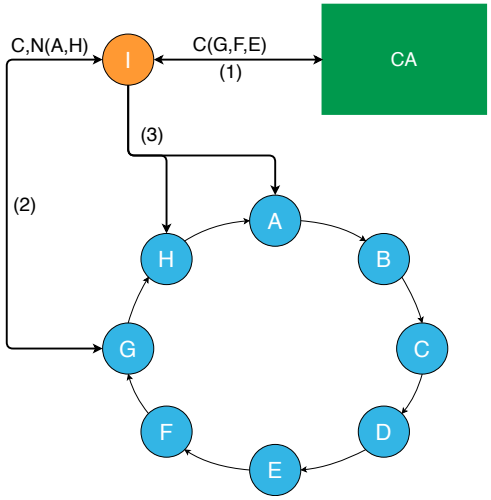


Figure 4.6: (1): *I* sends a certificate signing request to the CA. Receives certificates of participating peers *G, F, E*. (2): Tries to gossip with the closest neighbor *G*, which finds his neighbors since he has not seen him before. Returns certificate and notes of his appropriate neighbors, *A, H*. (3): *I* contacts his neighbors (*A, H*) and integrates into the network.

Peers already participating in the network are responsible for helping new members integrate into the network. A key consideration is therefore how much resources existing participants allocate to help integrate new peers. If we dedicate too much, adversaries could exploit this by repeatedly contacting participants, depleting resources. We want to evenly distributed resources among our neighbors, and occasionally help new peers integrate, but with a low amount of resource cost. Our solution is as follows; existing participants help newly joined peers integrate by finding their appropriate neighbors. More specifically, certificates and notes of their neighbors are returned in the gossip response. The flowchart of a gossip interaction is shown in Figure 4.7. We deem finding their neighbors a small task, which only includes determining their appropriate position in each ring.

As depicted in Figure 4.7, if we have observed a peer before (it is present in our full view) we reject its request. If an adversary gossips with a non-neighboring peer a, and a has not seen him before, a will find his neighbors and include the adversary's' note and certificate in his local gossip set. Thereby, other participants will eventually learn of the adversaries existence, thus, if he tries to contact another non-neighboring peer that has received his certificate, that peer will reject him.

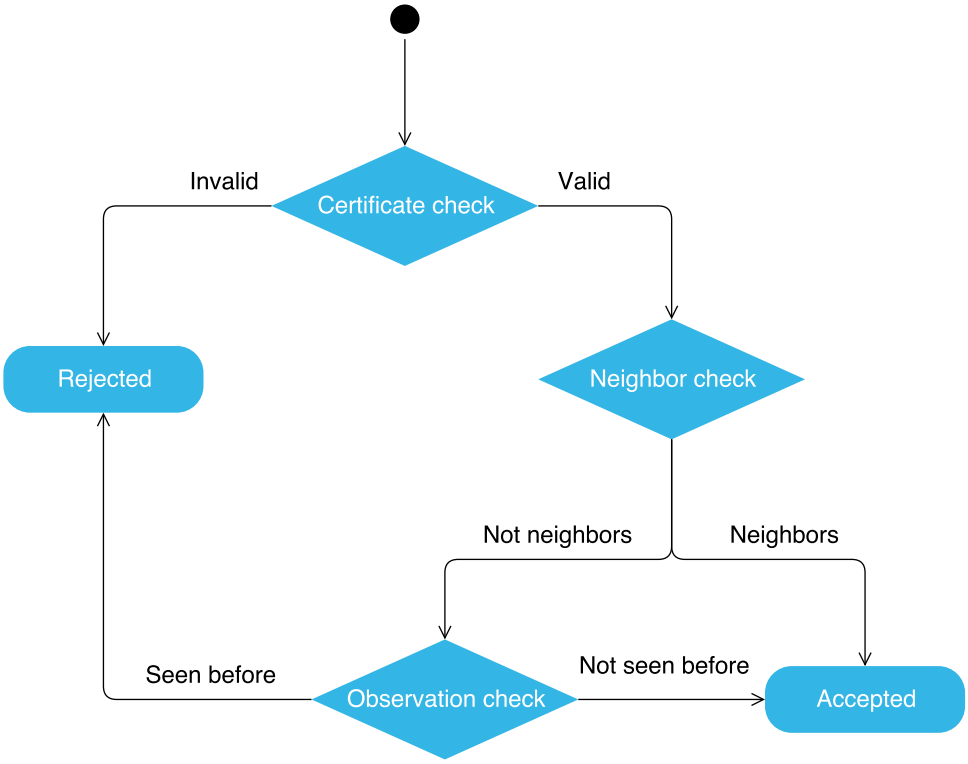


Figure 4.7: Flow of gossip interactions.

4.3.2 Rejoining after crashing

A peer might crash or temporarily become disconnected from the network. When detecting an unresponsive peer, accusations are created and disseminated. Upon receiving a valid accusation, a local timeout is started, associated with the accused participant. If the participant was already considered crashed or has not been observed before, the accusation is discarded. If the timer expires before receiving a rebuttal, the peer is removed from the live view and ring structures, but is still kept in the full view. When the peer becomes available again he contacts his neighbors, learns of the accusations concerning himself, rebuttals them and thus rejoins the network. However, there are scenarios where the rejoining peer will be rejected by his neighbors. While the peer was unavailable, new participants might have joined the network and become the new neighbors of the peer's previous neighbors, as shown in Figure 4.8.

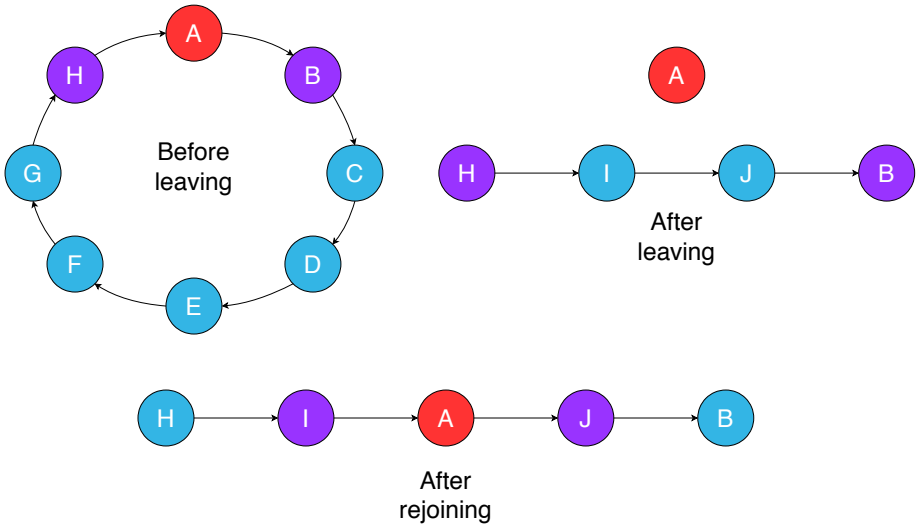


Figure 4.8: Process of rejoining the network after being crashed. *A* is not aware of his new neighbors after coming online again. Upon contacting his old neighbors, they inform him of the accusations associated with him. Subsequently, *A* rebuttals the accusation, learns of his new neighbors and rejoins the network.

Here, peer a becomes unavailable and does not learn of the accusation concerning himself. Before rejoining, peer i and j joins the network, and becomes the new neighbors of peer h and b. When peer a contacts h or b, he is rejected due to not being their neighbor and having previously been observed. Since a does not learn of his accusation, a rebuttal is not initiated. Also, a is unaware of i and j's existence since he has none to gossip with. Hence, he will remain crashed in other participants view, and unable to contact his neighbors.

We solve this issue by extending our observed check to include whether the contacting peer is accused. If so, we send back the accusations concerning him. Subsequently, the peer will issue a rebuttal, and at the next gossip interaction he will present his new note, which invalidates the accusations. Note that the contacting peer's note will always be evaluated, even if the interaction is rejected. The validation algorithm is shown in Code Listing 4.2.

Code Listing 4.2: Note evaluation.

```
func evaluateNote(n *note) {
    peer := getPeer(n.id)

    // Content validation.
    if !isMoreRecent(n, peer.note) || !validMask(n.mask)
       || !validSignature(n.signature) {
        return
    }

    // New note might be a rebuttal,
    // need to invalidate accusations and timeouts if so.
    if peer.isAccused() {
        peer.removeAccusations()
        peer.removeTimeouts()

        // Peer might already have been removed from our live view,
        // by a previous timeout expiring.
        if peer.isCrashed() {
            addLive(peer)
        }
    }
    // Always want to store the most recent note.
    peer.replaceNote(n)
}
```

Hence, when the peer retries to contact his former neighbors, the new note will invalidate previous accusations. After invalidation, the new note will be disseminated throughout the network, invalidating accusations at all other participants. As a result, the peer will be re-included into the live view and rings at all other participants.

4.3.3 Failure detector

Peers are assigned monitoring responsibilities to remove crashed members from our live view. Each participant monitors their successors by periodically pinging them, if a specified timeout is exceeded the ping fails, if this occurs more than a set limit the successor is considered dead. To prevent the forging of pong responses, each ping contains a random generated nonce value, pong responses contain the signature of this nonce. If the signature is not valid, the ping is considered failed. Thereby, malicious peers cannot forge ping responses to keep dead peers in the live view of honest peers.

Upon detecting a crashed peer, the accuser generates an accusation and adds it to the local gossip set. It will then be included in all the following gossip interactions and spread throughout the network. Only the direct predecessors of a given peer p can issue valid accusations for p. With high churn in the network, peers might disagree on the current system view, resulting in disagreement concerning the validity of accusations. However, participants will eventually converge to the same view when the churn ceases.

The ping protocol used in the original implementation [25] adopts the ping timeout according to previously recorded latencies. Hence, it does not rely on a set timeout shared across all connections. A hard timeout approach is sub-optimal since latencies will differentiate between connections in a distributed environment. Ifrit does not implement the ping protocol specified in [25], and relies on a set timeout for all connections. This a drawback of the implementation, and the ping protocol or another approach should be implemented in future work. Pinging in Ifrit is done over User Datagram Protocol (UDP) and relies on a set timeout, after a set amount of failed pings a node is considered to be crashed. As a result, Ifrit might generate more false accusations by not regulating the timeout per connection compared to the previous implementation [25].

Our failure detector is agnostic of the underlying pinging procedure, hence, any implementation can be provided. We implemented the pinging procedure using UDP due to its lower complexity compared to Transmission Control Protocol (TCP).

4.4 The Gossip service

Participants gossip through secure channels, where peers are securely identified by their signed certificate. By having a trusted CA and communicating over secure channels, Ifrit resists Sybil attacks [7]. All participants maintain a secure gossip channel with neighboring peers, as described in Section 4.2. After establishing a secure connection, gossip channels are reused until a change of neighbor or disconnection. In the case of disconnection, the neighbor will be re-dialed periodically until either he responds or a new neighbor replaces him. Peers periodically reconcile membership state with neighbors through these channels. When peers gossip, they pick a Ifrit ring in a round-robin fashion and gossip with both neighbors on that particular ring. Hence, with n rings, peers will have gossiped with all their neighbors after n rounds of gossip. Our gossip interval is a configuration variable, with the default set to 10 seconds. This is a rather aggressive approach, and we envisage that applications can configure this interval according to their needs. More frequent gossip results in quicker

convergence, but more bandwidth usage. We use Google Remote Procedure Call (GRPC) for all communication and connections over TLS.

Applications might want to disseminate their own gossip, which they typically would do on-top of any membership view. The standard approach in gossip protocols is to periodically select a random peer and reconcile information. With random peer selection, and other primitives, such protocols converge with high probability. For example, Bitcoin [2] does exactly this, periodically gossiping with neighboring peers about transactions, blocks and membership information. Ifrit enforces diverse gossip partners through its pseudo-random ring constructions, where in each ring, all participants are organized in a different order. Hence, periodically gossiping with neighbors is similar to selecting a random peer from a membership view. Therefore, we provide applications with a gossip service, where they can add gossip content which will periodically shared with neighboring peers. We effectively piggbyback application gossip on our own gossip interaction with Ifrit neighbors. When gossiping with neighboring peers, the provided message is included and propagated to the receiving application. Ifrit will include this message in each gossip interaction until the application either replaces or removes it. All messages and their responses are relayed through events that the application subscribes to, as shown in Figure 4.9.

The application is guaranteed that all messages received through gossip are sent by a neighboring peer in the Ifrit ring mesh. Peers that try to gossip with everyone for possibly devious purposes are rejected, as shown in Figure 4.7. All gossip related to Ifrit internals are still propagated and processed to maintain the membership, even if the application does not attach any additional gossip. We envisage that this service is beneficial for applications wanting to disseminate state in a distributed environment, similar to how Ifrit operates. As peers gossip at 10 second intervals by default, the provided message will be transferred over the network every 10 second. Therefore, applications should not include messages in the gossip service that are of significant size.

We do acknowledge that this service is possible to simulate through our messaging service or application messaging primitives. However, its purpose is to periodically exchange information with a restricted set of peers through a secure channel. From our gossiping rules, we can determine which peers are allowed to contact us. If applications were to enforce this through an alternative messaging scheme, they would have to keep track of all gossip rings and discard messages originating from non-neighbors. This would also require them to know how we manage our underlying rings, or enforce their own messaging rules. We argue that this service provides an ease-of-use alternative to implementing restrictive messaging protocols. Also, Ifrit already maintains gossip connections for its own purposes, so the only cost is an increased message

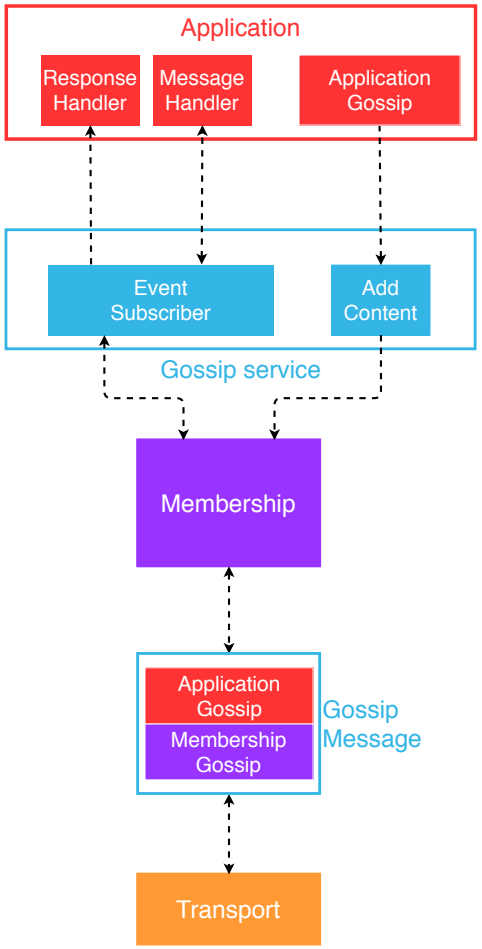


Figure 4.9: Application interaction with the gossip service.

size per gossip interaction, instead of additional connections in an application implemented scheme.

4.4.1 Gossip message content

As message size is one of the limiting properties of a gossip protocol [29], reducing gossip content is critical for performance. The original Fireflies [25] implementation supported set reconciliation [77] for gossip interactions, minimizing the amount of data transmitted over the network. Our previous Ifrit implementation did not support this, and instead transferred the entire gossip set in each interaction (all certifcates, notes, and accusations). We still do not support set reconciliation, but have changed our gossip approach to reduce network usage. Instead of sending everything, we send a set of all current

known peer id's and their current note epoch, as shown in Figure 4.10.

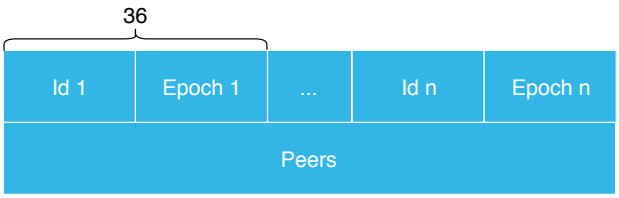


Figure 4.10: The contents of a gossip message, numbers refer to the sizes of each structure in bytes.

On the receiving side, we identify if the sender has any old notes and if they lack certain node id's. The response contains all notes we believe the sender has stale versions of, any certificates belonging to peers the sender did not know of, and all accusations as shown in Figure 4.11. Although corrupt peers could attack this design by repeatedly stating that they know of no other peers, this would result in the same network usage as our previous approach. Hence, our worst case scenario when under attack is equal to our previous design, while in a normal case scenario we reduce our network usage.

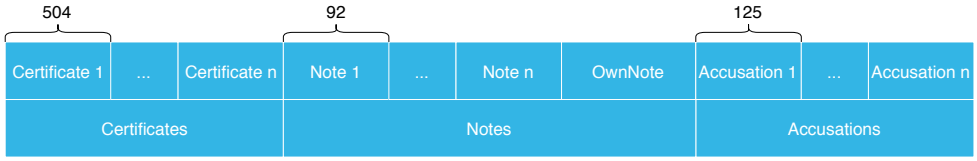


Figure 4.11: The contents of a gossip response, numbers refer to the sizes of each structure in bytes.

To further reduce network traffic, we compress all messages. GRPC supports compressing outbound messages, and uses GNU Zip (GZIP) by default. Compression presents a trade-off between Central Processing Unit (CPU) usage, memory usage, and network traffic. In theory, Ifrit is network bound, therefore we deem this trade-off acceptable. Also, our deployment environment has low quality network links, which further supports enabling compression.

4.5 The Messaging service

One of the benefits of full membership views is the possibility of point-to-point messaging, without any intermediate hops. With Ifrit's full membership view we provide a messaging service where applications can send their messages

directly to their destination. Addresses of other participating peers can be fetched from the membership service, and then be used to send direct messages. Applications can implement their own messaging scheme on-top of our membership service. However, we provide this service as an ease-of-use alternative to implementing a messaging scheme. Also, all messages are transferred over a secure connections TLS. With a full membership view and storing of all participants signed certificates, we can establish a TLS connection with every participant.

As network links are unpredictable and peers have different network environments, we implemented this as a asynchronous service. Applications subscribe to events such as incoming messages and responses, an example usage of sending a message is shown in Code Listing 4.3.

Code Listing 4.3: Message service.

```
1  func sendMessage(msg) *response {
2      // Retrieves addresses of all peers currently belived to be alive.
3      addrs := ifrit.Members()
4
5      // Pick random peer to send a message to.
6      addr := addrs[randIdx]
7
8      // Returns a synchronization primitive.
9      ch := ifrit.SendTo(addr, msg)
10
11     // Wait for response through the synchronization primitive.
12     resp <- ch
13
14     return resp
15 }
```

A key note here is that our messaging service bypasses Ifrit communication rules, as described in Section 4.2, messages can be sent to all participants and not just neighbors. We deem this a necessity to make the service applicable. If we did abide by Ifrit rules, the messaging service would become somewhat identical to our gossip service. However, when we allow applications to communicate with all participants, we need to consider connection management. If an application decides either by accident or on purpose, to message everyone in the network with a significant amount of participants, we cannot simply open connections to everyone. We employ a connection pool in this service, where only a fixed set of connections are allowed to operate concurrently. Hence, when applications send messages and exceed the max amount of connections, messages are queued until connections become available. Since the service is already asynchronous, this extension is feasible.

4.6 The Signature service

All peers are required to present a certificate containing their public key and network address that is signed by a trusted CA when contacting other participants. By maintaining a full membership view and storing public keys of all other participants, Ifrit is capable of verifying their signatures. However, applications must provide the Ifrit id of signer, such that Ifrit can verify the signature with the correct public key. Applications only need to include the id, which can be retrieved through our external interface, in the message they want to sign and send. We envisage that this service is beneficial for applications needing to verify content integrity, and we relieve them of disseminating public keys. For example, if the application uses either the gossip or message service, all outgoing messages can be signed and the received verified by the signature service.

One of the motivations for providing this service is that gossip protocols are inherently susceptible to data corruption [60]. Hence, if an application uses our gossip service to disseminate information, for example a vote table where each participant votes for a particular issue, malicious peers could manipulate other participants entries. By providing a signature service, applications sign their own entry, verify others and discard tampered information.

4.7 Certificate authority

Ifrit also supplies a basic Certificate Authority (CA) implementation responsible for signing participants' certificates. The CA exposes a single Hypertext Transfer Protocol (HTTP) endpoint for certificate signing requests, expecting the body to contain a *X.509* compliant certificate request. If the request body does not contain a valid certificate request, it is discarded. When signing certificates, the CA is responsible for generating the Ifrit id used by participants. As these ids have to be unique, the CA stores all previously generated ids to ensure uniqueness. Furthermore, the CA acts as a entry point into the network by piggybacking certificates of nodes already present in the network on certificate signing requests. This was done purely for simplicity and we consider the entry mechanism as an orthogonal field of research. A possibility would be to distribute the CA in similar manner as the DNS servers used for Bitcoin [29], acting as both a CA and DNS. We do not consider vulnerabilities or attacks concerning the certificate authority implementation.

As of now, our CA does not monitor network activity, hence, its unaware of which peers are still participating. As a result, if all of its known peers either leave or crash, it can no longer provide an entry point into the network. To

solve this, the CA could periodically query its set of known peers to ensure that it always has an updated view. Another approach could be to simply store all certificates at the CA, however, if the system grows to a significant size, both storage and determining who is alive becomes problematic.

4.8 Cryptography

Ifrit uses GO's standard library for all cryptography and certificate operations, both the CA and client implementations. Also, we adopt the same approach as the previous implementation [25] and use elliptic curve signatures due to its low signature length compared to Rivest–Shamir–Adleman (RSA) and Digital Signature Algorithm (DSA). As one of gossip protocol's limitations are bounded message sizes [60], and all gossiped data structures are signed, we deem this a desirable feature.

5

FireChain Consensus

This chapter will introduce FireChain's consensus component and its subcomponents. FireChain's consensus protocol is based on gossiping block propositions, adding them to the state component as participants agree upon the next block. Propositions converge over time so that eventually every correct member will (with a high probability) have seen all votes.

5.1 Consensus protocol

Time is divided into *epochs*. For each *epoch* members decide on the next block to commit to the chain. Hence, the epoch length in wall-clock time decides the block commit interval. In the current prototype, epochs are configured to be 10 minutes, split into 60 10 seconds gossip rounds. Leaving peers with 60 gossip rounds to agree upon the next block. In each *gossip round*, participants gossip k other peers in the system. At the end of an epoch participants commit their *favorite block* to their local chain. The favorite block of a participant is the block that received the majority of votes during an epoch, hence, its the most popular block. A superficial representation of the protocol is shown in Code Listing 5.1.

Code Listing 5.1: Consensus protocol.

```
// In each epoch we decide the next block.
for each epoch do {
    // We choose a favorite block in each round.
    for each gossipRound do {
        // Fetch k random peers from our view.
        kPeers := sampleRandomPeers()

        // Gossip with them, get their tables.
        tables := gossip(kPeers)

        // Update entries that has lower epoch numbers.
        roundTable := reconcileTables(tables)

        // Pick the most popular block for this round.
        // The block with the most votes becomes our favorite.
        pickFavoriteBlock(roundTable)
    }

    // After reaching agreement during gossip rounds we commit
    // our favorite block, and the process repeats.
    commitFavoriteBlock()
}
```

All participants maintain a *vote table*, containing one entry per participant in the network. Entries effectively represent each peer's vote for the next block. Each entry consists of: the peer's id, an epoch, the peer's favorite block, and a signature. Entries are signed by their respective creator to ensure that malicious participants cannot alter other peers' votes. The epoch field does not directly correlate to consensus epochs, but rather representing an increasing counter establishing the order of which block each peer favored. Hence, if peer n favors block a at epoch 5, but later on favors block b at epoch 6, n has changed favorite block from a to b during this timespan. Thereby when we are comparing entries, we know that the one with the highest epoch is the most recent vote from the respective peer.

In each gossip round, participants reconcile their vote table with k other peers. When reconciling tables, peers adopt all entries that have a higher epoch number compared to their entry. After reconciling tables, participants count votes for each block, and replaces their favorite block with most popular block in the table. Subsequently, incrementing their epoch counter, signaling a change in favorite block. If they chose the same block as they did in the previous round, they do not increment their epoch counters. In the event of a tie between several blocks, one is picked at random.

At the start of an epoch, each participant fills their local block with collected entries and sets it as their favorite block. After gossiping with other participants, peers will learn of other blocks and always vote for the most popular one. By

continuously gossiping tables, participants will eventually, with high probability, agree upon the next block. Epochs act as a global commit timer and signals all peers to commit their current favorite block, and start the consensus process for the next block. Peers ignore all votes that either have a invalid signature or has a different previous block compared to them, hence, ignoring votes originating from forks. We will introduce possible forking scenarios and how we resolve them later on.

5.2 Vote tables

Vote tables contain all participating peers, and each entries' structure is shown in Figure 5.1. Each peer uses their deployed Ifrit client's id as their vote table *id*

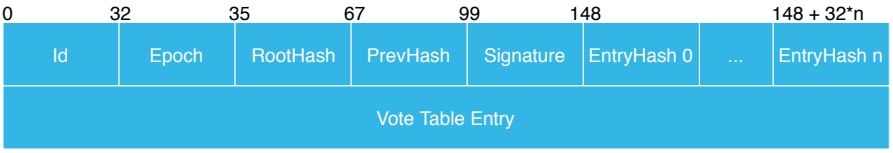


Figure 5.1: Structure of a vote table entry, numbers refer to their placement within the structure.

since they are already unique. To identify what block entries each favorite block consists of, we include hashes of all block entries in vote table entries. Thereby, peers can identify which block entries is missing from their local storage. The *roothash* field represents the merkle tree root hash of the block's entries, and *prevhash* represents the previous block's hash. By adding signatures, entries are tamper-proof. We utilize Ifrit's signature service to verify all signatures and to sign our own entry.

5.3 Gossiping vote tables & block entries

Vote tables and block entries are disseminated through Ifrit's gossip service. We refer to the message attached to the gossip service as our *state*, which will be transferred to our neighbors periodically by the service, subsequently reconciling their states. Upon a change in state, the gossip service is updated with the new state to be disseminated. A change in state does not append another message to the service, it replaces the previous one. We will now present our design process of what our state consists of in incremental steps, before finally showing our final approach.

Initially, our state contained the entire pending pool of block entries and our local vote table. However, each participant's pool could grow indefinitely since the pool had no size limit. As a result, gossip messages between participants would also grow indefinitely in size. To resolve this issue, a maximum size of the pending pool was imposed on participants, ensuring that gossip messages would never exceed the maximum memory pool size. However, this approach was not feasible since some entries might not be propagated throughout the network due to all participants having maxed out their memory pools. Peers need to have all block entries that could be included in the block to be chosen in the current epoch. We then instructed participants to continuously ask their neighbors for entries in their favorite block. However, with 300 participants and 1 MB blocks, each participant would have to receive and store 300 megabytes of data per epoch in a worst case scenario where no blocks had overlapping entries. To reduce network usage, participants only ask neighbors for entries they lack to complete their own favorite block, instead of all entries of everyone's favorite block. More specifically, each peer asks their neighbors for all entries in their missing pool, as explained in Subsection 3.2.4. As the network converges, all participants will eventually have the same favorite block, and everyone will have received its content. Hence, we will still fulfill our constraint that participants need to have all block entries of the block to be chosen in an epoch. At the point of convergence, everyone's missing pool will be empty and no further transfer of entries is necessary.

To determine which entries each block in the table consists of, each entry contains all entry hashes of the blocks content. The total size of a vote table entry with a block consisting of 10 entries would be 486 bytes. With 100 participants, the entire table would then be 48.6 KB. We initially added the entire table to the gossip service, where it would be sent over the network at each gossip interaction. Participants would then adopt specific entries as according to the protocol. The gossip service will by default compress all outgoing messages, reducing our network usage. However, we wanted to reduce it further. Firstly, we made a small change to the protocol, instead of incrementing our epoch number after each reconciliation, we only increment it after changing favorite block. We still add our entire table to the gossip service, however, each entry only consists of: id, epoch and the block's merkle root hash. Entry size is now reduced to 68 bytes, and with 100 participants amounts to 6.8 KB. On the receiving side, the participant checks whether the sender has any stale entries by checking epoch numbers and merkle hashes. If there any stale entries, their full versions (the one showed in Figure 5.1) are transferred back. Since the receiver does not adopt any entries, messages do not need to include signatures. They are transferred when stale entries are detected, likewise for entry hashes.

Our state consists of a reduced vote table, and the missing pool explained

in Subsection 3.2.4. We subscribe to incoming messages and responses by registering handlers in the gossip service, where our message handler is expected to return a response. Our registered event handlers pseudo code is shown in Code Listing 5.2, which will be invoked on each received message and response.

Code Listing 5.2: Gossip callbacks.

```
func handleGossip(msg *message) *response {
    // We fetch all vote table entries we think the sender has
    // stale versions of.
    staleVotes := checkTable(msg.table)

    // We return the requested block entries,
    // if we have them in local storage.
    blockEntries := getBlockEntries(msg.missingBlockEntries)

    resp := &response{
        blockEntries: blockEntries,
        table:        staleVotes,
    }

    return resp
}

// This callback will be invoked on each received response.
func handleGossipResponse(resp *response) {
    // We requested these in our inital request.
    for _, blockEntry := range resp.blockEntry {
        // Add all block entries that we are missing from our favorite block.
        if isMissing(blockEntry) {
            addBlockEntry(blockEntry)
        }
    }

    // The receiver has identified which votes we have
    // stale version of, now we reconcile them.
    reconcileVoteTable(resp.table)

    // If we update some vote table entries or
    // received some block entries missing from our favorite block,
    // we then update ifrit with our new state.
    if state.changed() {
        ifrit.SetGossipContent(state.serialize())
    }
}
```

In our initial design, when participants converged, they would still exchange full voting tables. With our optimization, we only transfer 68 bytes per participant after convergence. Even before convergence we only transfer stale entry values, reducing our network usage significantly. Additionally, participants would exchange significant amounts of block entries, even if they already had all block entries associated with their favorite block. As gossip protocols are inherently bound by message sizes [60], we deem this a significant improvement to our previous design.

As mentioned in Chapter 4, when using the gossip service, messages should not be of significant size. We observed slow peers with our initial design choices due to extensive network usage. Both due to responses growing indefinitely and large vote tables. By changing our gossip approach, we fetch a maximum of one block of entries and the size of vote table entries reduced significantly.

5.4 Resolving forks

Forks occur when one or more participants create a branch of blocks separate from the main chain. Bitcoin [2] resolves forks by participants always following the longest chain. Hence, the main chain will always be the set of peers with majority of computing power. However, it is not clear which branch is the main chain at point of forking, since they are equal in length. It takes time before the main chain out-paces other branches. This is one of the reasons Bitcoin developers recommend waiting until your transaction is 6 blocks deep before considering it permanently committed [29]. Another approach introduced by Ghost [58], proposes weighting each branch not just by its length, but by its subtrees. If a branch has significant amount of subtrees, it is more probable that its the main chain since there is probably more peers working on that branch.

As our consensus protocol is not based on POW, forks are not created by multiple miners finding the next block simultaneously. One or more forks are created if peers disagree at the end of an epoch, whether it happens by accident or due to malicious participants. During development, we found that a frequent cause of forks was peers temporarily becoming unavailable, not being able to contact or be contacted by other participants. Isolated peers would then progress their own local chain, oblivious to other peers in the network. Since they were alone, their own vote was enough to commit block after block. When becoming available again and rejoining the network, they would have a personal branch and would ignore other's blocks due to having different previous blocks. Likewise for other participants, rejecting the re-emerged peer's blocks due to different previous blocks.

To resolve forks, we utilize our full membership view combined with our vote tables. After rejoining the network, peers will learn of the current block proposals through reconciling vote tables. By examining updated vote tables, peers can identify if a majority of the network is on a different branch. Since each vote table entry contains the previous block, we deem that every participant with a different previous block is on a separate branch. If a majority of the network (over 50%) is on a different branch, peers contact a random participant in that majority and presents him with his local chain. We only send the

hashes of each block with no content, since the receiver only needs to identify where the fork occurred. The random participant then inspects where the fork occurred through his state component and sends back all the blocks beyond that. Subsequently, the receiver evaluates the proposed chain and replaces his own if valid. As we select a random participant in the majority, we might contact a byzantine or corrupt peer, which in turn could present his own secret fork. However, if we receive a secret fork that is not the main chain, we will simply detect this as another fork and the process is repeated. An obvious drawback of this approach is if the network is split into two equal partitions, hence, there is no majority. To improve on this, we could add a tie-breaker rule, such that all honest peers would at least follow the same chain. As peers in the majority might not be one of our neighbors, we cannot contact him directly through the gossip service. We also rely on this direct message to be through a secure channel. Peers utilizes Ifrit's messaging service to resolve forks through secure channels.

Another approach could be to contact a set amount of peers in the majority and confirm their chain representation from multiple sources before committing. This would, however, incur additional overhead and it is essentially what we do by repeating the process, just with a significantly better best-case performance. For example, if we were to contact 2/3 of the majority before committing, with a total network of 1000 and a majority of 900, that would be 600 messages. Instead we only send one message, and repeat the process if the received chain is a fraud attempt. This is only plausible due to our full membership view, with a partial view, we would not be able to accurately estimate what the majority of the network favored since we do not know its full size. Also, contacting someone in that majority, if not in our partial view, would have to travel several hops between peers to reach its destination. Increasing the chance of encountering a byzantine or corrupt peer at each hop.

An adversary attempting to create a fork in Bitcoin [2] could attempt to outpace the main chain if he is in possession of the majority of computing power. Hence, invalidating the main chain's progression in favor of his own blocks. Peers in FireChain do not produce POW to progress the chain as in Bitcoin, but collectively agree on the next block within a fixed time frame. Hence, a branch with only one honest member would progress at the same pace as the main chain. An adversary could produce blocks at a rate only bounded by the rate of which he can produce and disseminate them. However, these blocks would not be accepted by honest participants as they only commit blocks per fixed interval and commit the one which is most popular. This leads to the possibility of performing a Sybil attack [7], where adversaries allocate a significant amount of identities to gain influence and control a distributed system. However, Ifrit's membership service already solves this issue as explained in Chapter 4. Adversaries could, however, refuse to forward

gossip information or simply vote for random vote table entries that are not popular. Aiming to cause confusion among honest peers, preventing them from agreeing on the next block. To withstand such attacks we rely on our gossip converging at honest members.

Adversaries can simply pool their votes together and vote for their block, effectively controlling chain content. Content would still have to be valid, however, they could starve all other participants by not letting them commit their entries, rendering the system useless for all other peers. Honest participants will follow the most popular block, and if 30% of the network is corrupt, it is highly probable that their block will be chosen every round. We currently do not have any countermeasures, however, a POS hybrid approach could prevent this attack. If votes were weighted by participants current stake in the system, adversaries would at least have to own significant amounts of stake before they can control chain content. Current systems employing POS [8, 55] select a committee based on participants' stake in the system. The committee either collectively agrees upon the next block or elect a leader responsible for deciding the next block. We would not elect a full committee, and rather weight a stakeholder's vote higher than that of peers with no stake in the system. Alternatively, participants could only consider votes originating from stakeholders. With a full membership view, peers can identify who currently is in possession of the most stake. Non-stakeholders would effectively only be disseminating gossip.

Also, the adversary could create a fork originating arbitrarily in the distant past. From this fork he can produce enough blocks to match the length of the main chain. When at the same length and with the majority of votes, the adversary can re-write history by introducing his fork. As generating blocks is trivial without POW the attack is feasible. However, the adversary must control the network majority. We do not have an effective solution for when a majority of the network is corrupt. However, we envisage that corrupt participants could be detected and their certificates revoked, effectively removing them from the membership. This would have to be enforced by Ifrit, but could be detected by FireChain.

6

Evaluation

In this chapter we evaluate FireChain by investigating how many rounds of gossip are required for the system to converge. We run several experiments for different scenarios to investigate various aspects of FireChain's performance.

In some cases during our experiments, certain members became isolated or disconnected from the system, unable to send or receive messages from the other peers. Such partitioned members may create forks in the blockchain when reconnecting with system, and report back that they converged on their own branch in o rounds. We discard these measurements and ensure that they resolved their fork and continued on the main chain. Also, we discard measurements originating from forks created by subset of participants. We are only measuring the convergence of the main chain, and we ensure that participants eventually resolve their forks and rejoin the main chain.

6.1 Experimental platform & setup

All experiments and most of the testing and development were done on PlanetLab. PlanetLab[1] is a global research network maintained by several academic institutions, supporting development of new network services, consisting of 1353 nodes distributed across 717 sites. Sites are distributed worldwide including: France, Germany, Norway, Italy, and Spain. PlanetLab provides a distributed environment spread across the world with different network environments, which are the desired characteristics for testing our implementation.

With PlanetLab's high diversity in network environments and commodity hardware, we argue that our experiments are conducted in a real world setting where hosts regularly crash or disconnects, which we often experienced in development and testing of both FireChain and Ifrit. PlanetLab nodes have a minimum bandwidth requirement of 400 kilobits Per Second (KBPS). [2]

PlanetLab provided us with 48 nodes distributed all over Europe with high diversity in network environments. Experiments were orchestrated from our local machine at the UiT. Our CA was deployed on the same node in all experiments and was redeployed before each experiment. FireChain instances were deployed on all nodes followed by a 30 minute waiting period to ensure that all nodes had converged to the same view and start up traffic had ceased. As we only have 48 hosts, multiple FireChain instances were deployed evenly across all hosts. FireChain instances are also responsible for creating block content. The chain of events in each experiment deployment can be seen in Section 6.1.

Step 1 Deploy the CA.

Step 2 Deploy FireChain instances on all PlanetLab nodes.

Step 3 Wait 30 minutes to ensure the underlying Ifrit clients have converged.

Step 4 Start experiment by sending a start request to all instances.

Step 5 After each epoch, peers report back their convergence number for that epoch to our orchestrator at UiT.

Step 6 Let the chain progress 50 blocks (50 epochs).

Step 7 Shut down the CA and FireChain instances.

1. https://www.planet-lab.org/
2. https://www.planet-lab.org/node/222

We rely on all FireChain instances starting approximately at the same time, and deem the latency between each participant receiving the start request acceptable. Each instance records how many gossip rounds it uses to settle on each block. By collecting all instances' round number for each block we can determine their convergence time for each block by inspecting the highest round number. The highest round number recorded represents when the entire network agreed upon the block. We also ensure that each participant has converged to the same blockchain state by inspecting each participant's chain. For each experiment deployment, we measure the average amount of gossip rounds required for convergence.

We could have deployed our experiments on Amazon Web Services (AWS), providing us with significantly increased network connectivity and possibly better hardware. Or we could create a simulated environment, and add latencies as we see fit. However, by deploying on PlanetLab we are closer to a real-life scenario, where crashes and network outage are the default and not a rare occasion. Since PlanetLab is distributed across the entire globe, network latencies are high and connectivity low, which was experienced first-hand during development and experiments. We frequently received messages from PlanetLab support due to our excessive network transfers to low bandwidth destinations, notably this only occurred with our initial design and later ceased to occur.

6.2 Consensus experiment

We first investigate how many gossip rounds the blockchain protocol requires in order to converge. In blockchain systems, the time used for reaching consensus is decisive for performance, both in terms of throughput and latency. We commit blocks at fixed interval, hence, reaching consensus prior to the end of intervals does not increase performance, but indicates system stability. In our experiments, blocksize is set to 1 KB as we want to mainly test our consensus scheme, and not maximize throughput.

The results are shown in Figure 6.1, with the amount of participants on the x-axis and the amount of gossip rounds used to agree upon the next block on the y-axis. We observed 23 fork blocks during the entire experiment.

From our results, its clear that we are well within the 60 gossip round limit for convergence. This is due to our aggressive gossip rounds set to 10 seconds, and that we gossip with 2 peers in each round. We observe an increasing amount of rounds needed for convergence as we add more participants, which is to be expected. The graph does not follow a linear pattern, hence, gossip

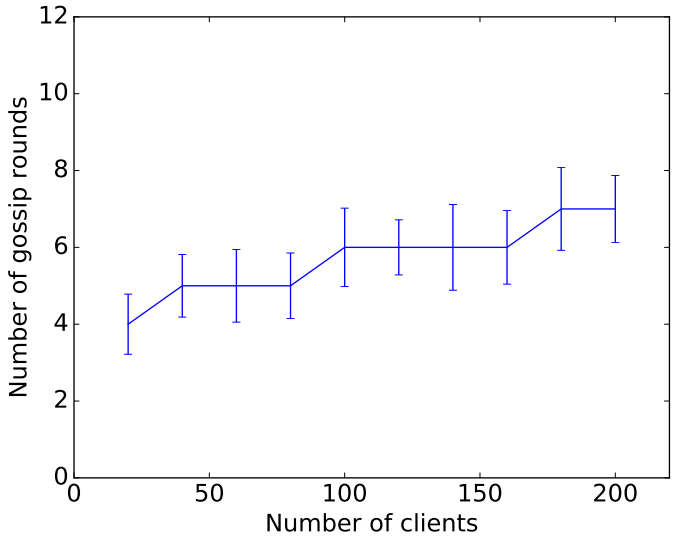


Figure 6.1: The average amount of gossip rounds required to reach consensus on blocks, with a 10 minute commit interval. Error bars show the 95 percentile.

dissemination of our vote tables are not linearly correlated with our convergence time. FireChain scales well to 200 participants, and by the trend of the graph, probably scales well beyond that. As we only has access to 48 physical hosts, we did not conduct experiment with even more participants, we leave this to future work.

As our epoch length is set to 10 minutes, and with 200 participants we currently use around 6-8 gossip rounds (slightly above a minute) to agree upon the next block, we could possibly reduce our epoch length. With rounds at 10 seconds, epochs could be pushed to around 2 minutes. Although, with a shorter time frame comes an increased chance of creating forks due to higher probability of peers disagreeing. Essentially presenting a trade-off between system stability and commit latency. With more forks, participants will more frequently disagree on the current state of the distributed ledger.

Throughout the entire experiment we observed 23 fork blocks, all of which only had either 1 or 2 votes. When peers temporarily become unavailable due to loss of network connectivity, they will simply progress their own local chain with their single vote. After rejoining the network and receiving update vote tables, they will detect a fork and resolve their branch with a random member of the majority. Hence, such forks live for the duration of the peer's network outage.

6.3 Block commit interval experiment

In the next experiment, we want to investigate if it is feasible to shorten epoch length. From the experiment described above, we observed that participants were able to agree upon a block on average before two minutes. This experiment will therefore explore how well FireChain perform with 2 minute epochs. However, with shorter epochs the probability of forks increases. We will either observe similar results to our first experiment or participants will not be able to agree within 2 minutes and diverge. As we have a significant shorter time frame to reach consensus, we also expect to observe more forks. Blocksize is set to 1 KB as we want to mainly test our consensus scheme, and not maximize throughput.

The results is shown in Figure 6.2, with the amount of participants on the x-axis and the amount of gossip rounds used to agree upon the next block on the y-axis. We observed 217 fork blocks during the experiment.

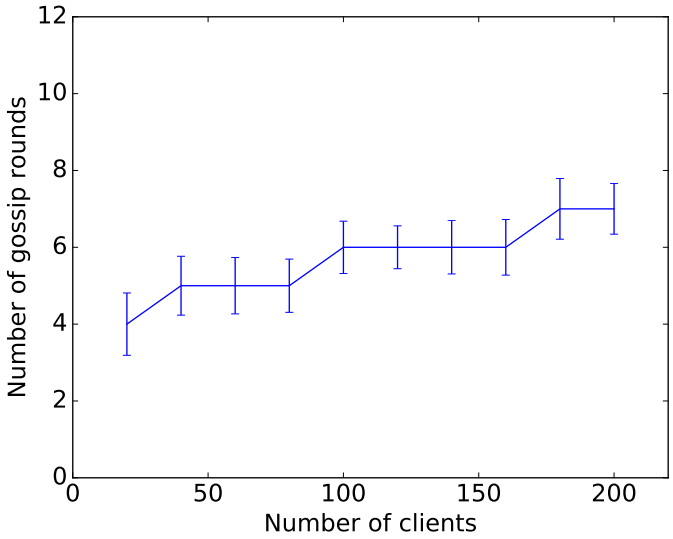


Figure 6.2: The average amount of gossip rounds to reach consensus on blocks, with a 2 minute commit interval. Error bars show the 95 percentile.

From our results, it is clear that participants are still able to agree upon blocks. We see an almost identical graph compared to experiment 1, hence, our system is still functioning as expected. However, we are creating significantly more forks compared to our first experiment, which is to be expected. This is due to the shorter time span in which peers have to agree upon blocks. If a peer becomes unavailable for two minutes, he has effectively missed an entire block selection.

We also observed longer lived forks, where separate vote partitions formed, these forks could live across several blocks. However, they were eventually resolved by our fork resolving scheme. This essentially presents a trade-off between fork frequency and block commit time. By committing blocks more frequently we reduce commit latencies, but we create more forks resulting increased system instability. Although we have showed the feasibility of running two minute commit intervals, a somewhat higher interval time might be sensible to account for various transient delays in the network.

With more forks, entries are more probable to not be permanently committed due to being on a fork. Bitcoin [2] advises a 6 block rule with 10 minute block commit latencies, resulting in a total of 60 minutes latencies. We do not have POW, hence, the main chain cannot be out-paced. Peers only have to be concerned about being apart of the majority. Hence, if a majority of all participants voted for a block, it is permanently committed. Unless an adversary gains control of the network majority, he is then capable of altering previous blocks as he holds a majority of votes. At the time of block committing, participants could prematurely detect that they are creating a fork by inspecting the amount of votes their favorite block has. If it is less than the majority, it is highly probable that the favorite block will become a fork.

An alternative optimistic approach that could improve performance, would be to allow agreement within an epoch. This would however require substantial change to the consensus mechanisms and require some changes to our fork resolving scheme. Instead of enforcing fixed time intervals for epochs, we could commit as we gained a majority for a block. Effectively committing whenever the network majority agree upon the next block, instead of waiting until the end of the current epoch. For example, if a network of 250 participants all voted for block b at gossip round 8, they would effectively have to wait until the end of the epoch (gossip round 60) to commit the block. We could allow the network to commit blocks that are in practice agreed upon prior to the end of the current epoch, thereby committing agreed upon blocks earlier. Participants that crash or are slow will simply fall behind have to catch up later. This approach introduces more instability and would require more investigation to determine if its a valid solution. However, we would eliminate our dead period between agreeing upon a block and waiting for the epoch to end.

6.4 Passive attack experiment

In this experiment we want to investigate how the system behaves when under attack. We instruct 30% of participants to mount a passive attack, and still measure gossip rounds required for convergence as in the previous experiments.

The goal here is to see if our system is capable of operating while under attack. Passive attackers still participate in the underlying Ifrit network, but do not participate in our consensus protocol and does not forward any FireChain gossip. Hence, attackers are attempting to disrupt the convergence process of honest participants. Attackers still deploy Ifrit clients, but do not subscribe to events in either the gossip or message service, they will therefore not receive any gossip or messages concerning FireChain consensus.

The results is shown in Figure 6.3, with the amount of participants on the x-axis and the amount of gossip rounds used to agree upon the next block on the y-axis. We observed 34 fork blocks during the experiment.

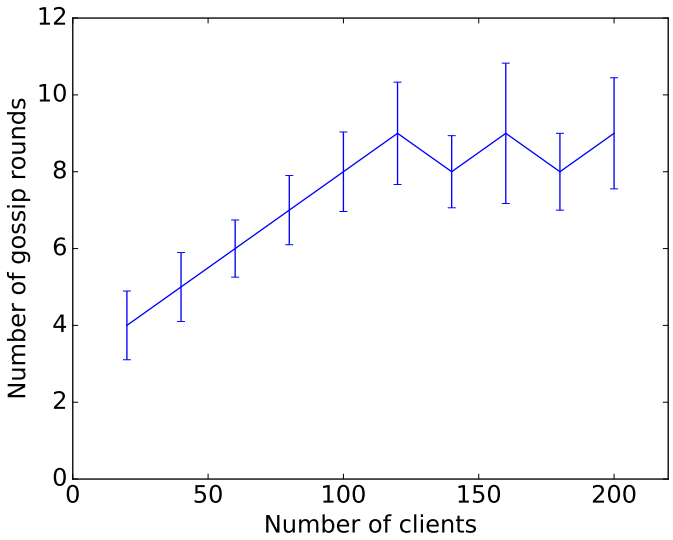


Figure 6.3: The average amount of gossip rounds to reach consensus on blocks, while under a passive attack. Error bars show the 95 percentile.

From the results we see that participants are still able to agree upon blocks despite the passive attack. We observe an increased amount of gossip rounds needed to reach consensus, but that is to be expected. The graph follows a clear linear pattern up until 120 clients, and afterwards varies. One possible reason for the difference in behavior is how attackers are positioned within Ifrit's ring mesh structure. Since each deployment produces its own pseudo-random ring mesh, attackers might be positioned in an inefficient attack manner. For example, if a high percentage of attackers are neighbors with the same honest participants, they can fully exclude them from participating in the consensus protocol. On the other hand if attackers are evenly distributed, honest participants will still receive consensus gossip, but in lower quantities. Additionally, if attackers are

neighbors they effectively waste an attack opportunity by affecting less honest participants' gossip patterns.

Surprisingly, we do not generate a significant amount of fork blocks. In our first experiment we generated 23 fork blocks, while under attack we produce 34. As attackers are effectively slowing down consensus progress, we expected even more forks to be created due to disagreement. This might be related to the same scenarios as we explained previously, however, as fork blocks are accumulated throughout all experiment deployments, its seems highly unlikely. We attribute this attack resiliences to Ifrit's gossip service. With Ifrit's ring mesh, participants are highly probable to have at least one honest neighbor and will thereby receive consensus gossip.

6.5 Block size experiment

In this experiment we want to test FireChain with 10 KB blocks and see if participants still reach agreement within epoch time frames. Previous experiments were conducted with 1 KB blocks, mainly to test our consensus mechanism. As disseminating larger blocks requires more bandwidth and time, participants might not reach agreement within epochs. Also, our vote tables will be significantly affected due to blocks having more entries, since we store block entry hashes in each vote table entry. Thereby, not only will block dissemination be affected, but also our vote tables that control our consensus mechanism.

The results is shown in Figure 6.4, with the amount of participants on the x-axis and the amount of gossip rounds used to agree upon the next block on the y-axis. We observed 34 fork blocks during the experiment.

From our results we see that participants are still able to agree upon blocks, and follows a similar scaling pattern as our previous experiments. However, we observe that at 20 and especially 100 participants, our results indicate system instability. After closer inspection we discovered that PlanetLab had capped bandwidth usage of some of our hosts due to excessive network traffic. Thereby resulting in slower dissemination of vote tables and blocks for some hosts, subsequently slowing the down our consensus protocol. As we extract the highest gossip round number per block, the slowest participants will determine when the system agreed upon blocks. Thus, we argue that the instability at 20 and 100 participants were due to some hosts having the bandwidth capped. Although, it is interesting that our bandwidth was not capped with 200 participants, as we use more bandwidth the more peers present in the network.

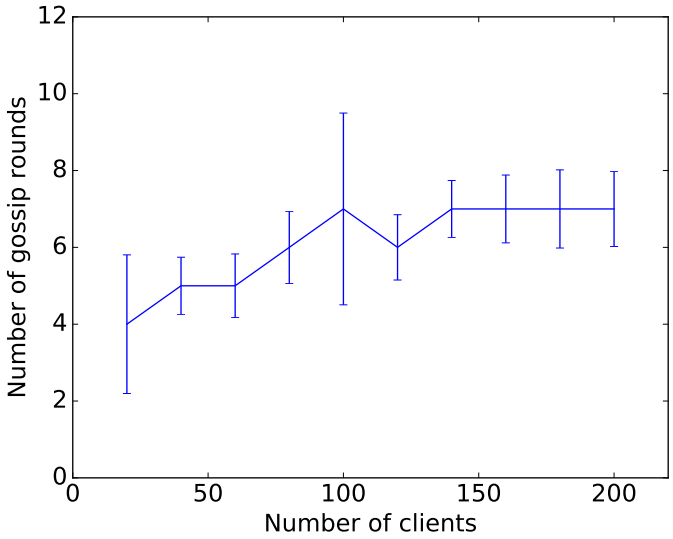


Figure 6.4: The average amount of gossip rounds to reach consensus on 10 KB blocks. Error bars show the 95 percentile.

We initially intended to set our blocksize to 1 MB, as in Bitcoin. However, after testing our implementation on PlanetLab, we discovered that nodes had insufficient bandwidth to disseminate blocks of that size. This was mainly due to our gossip approach, where all entry hashes associated with a block are stored in vote tables entries. Hence, as we increase block sizes, our vote table entries contain more hashes, subsequently consuming more bandwidth. Resolving forks and disseminating blocks also consumes more bandwidth with higher block sizes. Also, the minimum bandwidth requirement of PlanetLab nodes are 400 KBPS which we deem quite low, and has a maximum bandwidth usage limit of 10 Gigabyte (GB) per day.

If we separate our voting mechanism from our block entry dissemination, we could effectively agree upon blocks of arbitrary size. More specifically, we could remove block entry hashes from vote table entries, only identifying blocks from their root hash and previous hash. However, we would need another approach for coupling blocks with their respective content. Participants could potentially start disseminating block content after receiving sufficiently amounts of votes for their block, thereby shrinking the size of our vote tables significantly, and eliminating unnecessary block entry dissemination. With this approach participants could agree upon blocks without having received all of its content, possibly committing blocks with invalid content, which subsequently increases fork frequency. This approach or other alternatives should be explored in future work to further improve performance.

/7 Conclusion

Existing blockchain systems based on Proof-Of-Work (POW) consensus require immense amounts of energy to meet their safety and liveness properties. Systems based on classical Byzantine Fault-Tolerance (BFT) consensus avoids excessive energy consumption, but does not scale to the same extent, and has closed membership. Both POW and BFT systems often employ partial membership views. We deem this a disadvantage as this requires messages to be routed over multiple hops before reaching their destination.

Our goal was to build a blockchain based on another consensus approach, with a full membership view, and without the computational costs of POW. To facilitate our consensus approach, we designed and implemented a Byzantine fault-tolerant gossip and membership service, namely Ifrit. In general, we argue that Ifrit provides a beneficial membership service for blockchain systems. POW chains can utilize Ifrit without a CA and take advantage of the full membership view. POS and BFT chains can deploy Ifrit with a CA for a full Sybil resistant membership view.

With Ifrit's services, we designed and implemented FireChain, a blockchain based on an alternative consensus model, namely gossip that converges with high probability. From our experiments, we have shown that FireChain scales to 200 members. We were only limited by the size of our testbed, and we are confident that FireChain can scale even further.

7.1 Concluding remarks

By designing, implementing, and evaluating FireChain, we have shown the feasibility of building blockchain systems on a gossip consensus mechanism and full membership views. We successfully implemented a BFT gossip and membership service, namely Ifrit, based on the Fireflies protocol. Subsequently, we used Ifrit as FireChain's communication substrate. From our results we have shown the feasibility and scalability of FireChain, which operates without POW, hence, consuming low amounts of energy.

7.2 Future work

Future work consists of two main points; entry content and storage. As of now, entries contain random data, we envisage that FireChain can provide a generic interface where applications can define how transactions are validated, processed etc. Thereby, supporting any application specified transaction or data fulfilling our interface. We deem adding storage a trivial matter. Also, our fork resolving scheme has not been formally verified for correctness, and there might be attack vectors that we have not addressed.

Our experiments were conducted with a total of 48 physical hosts, where multiple clients were deployed on each host. In future work, larger scale should be done, conducting experiments at the scale of the Bitcoin network would be desirable. As we are confident in the scalability of FireChain, experiments with over 3000 physical hosts would be an impressive feat. Also, conducting experiments in a high performance environment would be an interesting comparison to our current environment on PlanetLab, where nodes have a minimum bandwidth requirement of 400 KBPS.

Bibliography

[1] M. Swan, *Blockchain: Blueprint for a new economy*. " O'Reilly Media, Inc.", 2015.

[2] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008.

[3] M. Vukolić, "The quest for scalable blockchain fabric: Proof-of-work vs. bft replication," in *International Workshop on Open Problems in Network Security*, pp. 112–125, Springer, 2015.

[4] E. Tedeschi, H. D. Johansen, and D. Johansen, "Trading network performance for cash in the bitcoin blockchain," in *Proceedings of the 8th International Conference on Cloud Computing and Services Science*, pp. 643–650, 2018.

[5] E. Tedeschi, H. D. Johansen, and D. Johansen, "Trading network performance for cash in the bitcoin blockchain," in *Proceedings of the 8th International Conference on Cloud Computing and Services Science, CLOSER 2018, Funchal, Madeira, Portugal, March 19-21, 2018.*, pp. 643–650, 2018.

[6] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum Project Yellow Paper*, vol. 151, pp. 1–32, 2014.

[7] J. R. Douceur, "The sybil attack," in *International Workshop on Peer-to-Peer Systems*, pp. 251–260, Springer, 2002.

[8] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich, "Algorand: Scaling byzantine agreements for cryptocurrencies," in *Proceedings of the 26th Symposium on Operating Systems Principles*, pp. 51–68, ACM, 2017.

[9] K. Croman, C. Decker, I. Eyal, A. E. Gencer, A. Juels, A. Kosba, A. Miller, P. Saxena, E. Shi, E. G. Sirer, *et al.*, "On scaling decentralized blockchains," in *International Conference on Financial Cryptography and Data Security*, pp. 106–125, Springer, 2016.

[10] M. Castro, B. Liskov, *et al.*, "Practical byzantine fault tolerance," in *OSDI*, vol. 99, pp. 173–186, 1999.

[11] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Computing Surveys (CSUR)*, vol. 22, no. 4, pp. 299–319, 1990.

[12] L. Lamport *et al.*, "Paxos made simple," *ACM Sigact News*, vol. 32, no. 4, pp. 18–25, 2001.

[13] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 4, no. 3, pp. 382–401, 1982.

[14] J. Kwon, "Tendermint: Consensus without mining," *Retrieved May*, vol. 18, p. 2017, 2014.

[15] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, *et al.*, "Hyperledger fabric: a distributed operating system for permissioned blockchains," *arXiv preprint arXiv:1801.10228*, 2018.

[16] G. Greenspan, "Multichain private blockchain, white paper," *URl: http://www. multichain. com/download/MultiChain-White-Paper. pdf*, 2015.

[17] J. Morgan, "Quorum white paper," *New York: JP Morgan Chase*, 2016.

[18] S. Voulgaris, D. Gavidia, and M. Van Steen, "Cyclon: Inexpensive membership management for unstructured p2p overlays," *Journal of Network and Systems Management*, vol. 13, no. 2, pp. 197–217, 2005.

[19] A. J. Ganesh, A.-M. Kermarrec, and L. Massoulié, "Scamp: Peer-to-peer lightweight membership service for large-scale group communication," in *International Workshop on Networked Group Communication*, pp. 44–55, Springer, 2001.

[20] R. Van Renesse, K. P. Birman, and S. Maffeis, "Horus: A flexible group communication system," *Communications of the ACM*, vol. 39, no. 4, pp. 76–83, 1996.

[21] A. Rowstron and P. Druschel, "Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems," in *IFIP/ACM International Conference on Distributed Systems Platforms and Open Dis-*

tributed Processing, pp. 329–350, Springer, 2001.

[22] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," *ACM SIGCOMM Computer Communication Review*, vol. 31, no. 4, pp. 149–160, 2001.

[23] A. Gupta, B. Liskov, R. Rodrigues, *et al.*, "One hop lookups for peer-to-peer overlays.," in *HotOS*, pp. 7–12, 2003.

[24] A. J. Ganesh, A.-M. Kermarrec, and L. Massoulié, "Peer-to-peer membership management for gossip-based protocols," *IEEE transactions on computers*, vol. 52, no. 2, pp. 139–149, 2003.

[25] H. D. Johansen, R. V. Renesse, Y. Vigfusson, and D. Johansen, "Fireflies: A secure and scalable membership and gossip service," *ACM Transactions on Computer Systems (TOCS)*, vol. 33, no. 2, p. 5, 2015.

[26] H. Johansen, A. Allavena, and R. Van Renesse, "Fireflies: scalable support for intrusion-tolerant network overlays," in *ACM SIGOPS Operating Systems Review*, vol. 40, pp. 3–13, ACM, 2006.

[27] M. Ripeanu, "Peer-to-peer architecture case study: Gnutella network," in *Peer-to-Peer Computing, 2001. Proceedings. First International Conference on*, pp. 99–100, IEEE, 2001.

[28] H. Johansen and D. Johansen, "Improving object search using hints, gossip, and supernodes," in *the 21st IEEE Symposium on Reliable Distributed Systems: Workshop on Reliable Peer-to-Peer Distributed Systems*, pp. 336–340, IEEE, Oct. 2002.

[29] C. Decker and R. Wattenhofer, "Information propagation in the bitcoin network," in *Peer-to-Peer Computing (P2P), 2013 IEEE Thirteenth International Conference on*, pp. 1–10, IEEE, 2013.

[30] R. van Renesse, "A blockchain based on gossip?–a position paper,"

[31] P. T. Eugster, R. Guerraoui, A.-M. Kermarrec, and L. Massoulié, "Epidemic information dissemination in distributed systems," *Computer*, vol. 37, no. 5, pp. 60–67, 2004.

[32] D. E. Comer, D. Gries, M. C. Mulder, A. Tucker, A. J. Turner, P. R. Young, and P. J. Denning, "Computing as a discipline," *Communications of the ACM*, vol. 32, no. 1, pp. 9–23, 1989.

[33] D. Johansen, K. Marzullo, and K. Lauvset, "An approach towards an agent computing environment," in *Proceedings. 19th IEEE International Conference on Distributed Computing Systems. Workshops on Electronic Commerce and Web-based Applications. Middleware*, pp. 78–83, 1999.

[34] D. Johansen, H. Johansen, and R. van Renesse, "Environment mobility: Moving the desktop around," in *Proceedings of the 2Nd Workshop on Middleware for Pervasive and Ad-hoc Computing*, MPAC '04, (New York, NY, USA), pp. 150–154, ACM, 2004.

[35] D. Johansen, R. van Renesse, and F. B. Schneider, "Operating system support for mobile agents," in *Proceedings 5th Workshop on Hot Topics in Operating Systems (HotOS-V)*, pp. 42–45, May 1995.

[36] L. Brenna and D. Johansen, "Configuring push-based web services," in *International Conference on Next Generation Web Services Practices (NWeSP'05)*, pp. 6 pp.–, Aug 2005.

[37] D. Johansen, R. van Renesse, and F. B. Schneider, *WAIF:Web of Asynchronous Information Filters*, pp. 81–86. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003.

[38] R. v. Renesse, H. Johansen, N. Naigaonkar, and D. Johansen, "Secure abstraction with code capabilities," in *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pp. 542–546, Feb 2013.

[39] H. Johansen, D. Johansen, and R. van Renesse, "Firepatch: Secure and time-critical dissemination of software patches," in *New Approaches for Security, Privacy and Trust in Complex Environments* (H. Venter, M. Eloff, L. Labuschagne, J. Eloff, and R. von Solms, eds.), (Boston, MA), pp. 373–384, Springer US, 2007.

[40] G. Hartvigsen and D. Johansen, "Co-operation in a distributed artificial intelligence environment—the stormcast application," *Engineering Applications of Artificial Intelligence*, vol. 3, no. 3, pp. 229 – 237, 1990.

[41] A. T. Gjerdrum, R. Pettersen, H. D. Johansen, and D. Johansen, "Performance of trusted computing in cloud infrastructures with intel sgx," in *Proceedings of the 7th International Conference on Cloud Computing and Services Science. Porto, Portugal: SCITEPRESS*, pp. 696–703, 2017.

[42] H. D. Johansen, E. Birrell, R. van Renesse, F. B. Schneider, M. Stenhaug, and D. Johansen, "Enforcing privacy policies with meta-code," in *Proceed-*

ings of the 6th Asia-Pacific Workshop on Systems, APSys '15, (New York, NY, USA), pp. 16:1–16:7, ACM, 2015.

[43] P. Halvorsen, S. Sægrov, A. Mortensen, D. K. Kristensen, A. Eichhorn, M. Stenhaug, S. Dahl, H. K. Stensland, V. R. Gaddam, C. Griwodz, *et al.*, "Bagadus: an integrated system for arena sports analytics: a soccer case study," in *Proceedings of the 4th ACM Multimedia Systems Conference*, pp. 48–59, ACM, 2013.

[44] H. K. Stensland, V. R. Gaddam, M. Tennøe, E. Helgedagsrud, M. Næss, H. K. Alstad, A. Mortensen, R. Langseth, S. Ljødal, O. Landsverk, C. Griwodz, P. Halvorsen, M. Stenhaug, and D. Johansen, "Bagadus: An integrated real-time system for soccer analytics," *ACM Trans. Multimedia Comput. Commun. Appl.*, vol. 10, pp. 14:1–14:21, Jan. 2014.

[45] S. Sægrov, A. Eichhorn, J. Emerslund, H. K. Stensland, C. Griwodz, D. Johansen, and P. Halvorsen, "Bagadus an integrated system for soccer analysis," in *Distributed Smart Cameras (ICDSC), 2012 Sixth International Conference on*, pp. 1–2, IEEE, 2012.

[46] D. Johansen, M. Stenhaug, R. B. Hansen, A. Christensen, and P.-M. Høgmo, "Muithu: Smaller footprint, potentially larger imprint," in *Digital Information Management (ICDIM), 2012 Seventh International Conference on*, pp. 205–214, IEEE, 2012.

[47] M. Stenhaug, Y. Yang, C. Gurrin, and D. Johansen, "Muithu: A touch-based annotation interface for activity logging in the norwegian premier league," in *International Conference on Multimedia Modeling*, pp. 365–368, Springer, 2014.

[48] D. Johansen, P. Halvorsen, H. Johansen, H. Riiser, C. Gurrin, B. Olstad, C. Griwodz, Å. Kvalnes, J. Hurley, and T. Kupka, "Search-based composition, streaming and playback of video archive content," *Multimedia Tools and Applications*, vol. 61, pp. 419–445, Nov 2012.

[49] H. Johansen, A. Allavena, and R. van Renesse, "Fireflies: Scalable support for intrusion-tolerant network overlays," *ACM SIGOPS Operation System Review: Proceedings of the 1st ACM EuroSys Conferance*, vol. 40, Oct. 2006.

[50] D. Dolev, E. N. Hoch, and R. van Renesse, "Self-stabilizing and byzantine-tolerant overlay network," in *Principles of Distributed Systems* (E. Tovar, P. Tsigas, and H. Fouchal, eds.), (Berlin, Heidelberg), pp. 343–357, Springer Berlin Heidelberg, 2007.

[51] K. J. O'Dwyer and D. Malone, "Bitcoin mining and its energy footprint," 2014.

[52] I. Bentov, A. Gabizon, and A. Mizrahi, "Cryptocurrencies without proof of work," in *International Conference on Financial Cryptography and Data Security*, pp. 142–157, Springer, 2016.

[53] P. Vasin, "Blackcoin's proof-of-stake protocol v2," *URL: https://blackcoin. co/blackcoin-pos-protocol-v2-whitepaper. pdf*, 2014.

[54] S. King and S. Nadal, "Ppcoin: Peer-to-peer crypto-currency with proof-of-stake," *self-published paper, August*, vol. 19, 2012.

[55] A. Kiayias, A. Russell, B. David, and R. Oliynykov, "Ouroboros: A provably secure proof-of-stake blockchain protocol," in *Annual International Cryptology Conference*, pp. 357–388, Springer, 2017.

[56] V. Buterin and V. Griffith, "Casper the friendly finality gadget," *arXiv preprint arXiv:1710.09437*, 2017.

[57] A. Miller and J. J. LaViola Jr, "Anonymous byzantine consensus from moderately-hard puzzles: A model for bitcoin," *Available on line: http://nakamotoinstitute. org/research/anonymous-byzantine-consensus*, 2014.

[58] Y. Sompolinsky and A. Zohar, "Secure high-rate transaction processing in bitcoin," in *International Conference on Financial Cryptography and Data Security*, pp. 507–527, Springer, 2015.

[59] N. T. Bailey *et al.*, *The mathematical theory of infectious diseases and its applications*. Charles Griffin & Company Ltd, 5a Crendon Street, High Wycombe, Bucks HP13 6LE., 1975.

[60] K. Birman, "The promise, and limitations, of gossip protocols," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 5, pp. 8–13, 2007.

[61] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong, "Tag: A tiny aggregation service for ad-hoc sensor networks," *ACM SIGOPS Operating Systems Review*, vol. 36, no. SI, pp. 131–146, 2002.

[62] D. Kempe, A. Dobra, and J. Gehrke, "Gossip-based computation of aggregate information," in *Foundations of Computer Science, 2003. Proceedings. 44th Annual IEEE Symposium on*, pp. 482–491, IEEE, 2003.

[63] R. Van Renesse, K. P. Birman, and W. Vogels, "Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining," *ACM transactions on computer systems (TOCS)*, vol. 21, no. 2, pp. 164–206, 2003.

[64] I. Eyal, A. E. Gencer, E. G. Sirer, and R. Van Renesse, "Bitcoin-ng: A scalable blockchain protocol.," in *NSDI*, pp. 45–59, 2016.

[65] Y. Hirai, "Defining the ethereum virtual machine for interactive theorem provers," in *International Conference on Financial Cryptography and Data Security*, pp. 520–535, Springer, 2017.

[66] C. Dannen, *Introducing Ethereum and Solidity*. Springer, 2017.

[67] Y. Lewenberg, Y. Sompolinsky, and A. Zohar, "Inclusive block chain protocols," in *International Conference on Financial Cryptography and Data Security*, pp. 528–547, Springer, 2015.

[68] Y. Sompolinsky, Y. Lewenberg, and A. Zohar, "Spectre: A fast and scalable cryptocurrency protocol.," *IACR Cryptology ePrint Archive*, vol. 2016, p. 1159, 2016.

[69] I. Bentov, P. Hubácek, T. Moran, and A. Nadler, "Tortoise and hares consensus: the meshcash framework for incentive-compatible, scalable cryptocurrencies.," *IACR Cryptology ePrint Archive*, vol. 2017, p. 300, 2017.

[70] S. Micali, "Algorand: The efficient and democratic ledger," *arXiv preprint arXiv:1607.01341*, 2016.

[71] S. Micali, M. Rabin, and S. Vadhan, "Verifiable random functions," in *Foundations of Computer Science, 1999. 40th Annual Symposium on*, pp. 120–130, IEEE, 1999.

[72] C. Cachin, "Architecture of the hyperledger blockchain fabric," in *Workshop on Distributed Cryptocurrencies and Consensus Ledgers*, 2016.

[73] E. Androulaki, C. Cachin, A. De Caro, A. Kind, and M. Osborne, "Cryptography and protocols in hyperledger fabric," in *Real-World Cryptography Conference*, 2017.

[74] J. Sousa, A. Bessani, and M. Vukolić, "A byzantine fault-tolerant ordering service for the hyperledger fabric blockchain platform," *arXiv preprint arXiv:1709.06921*, 2017.

[75] E. K. Kogias, P. Jovanovic, N. Gailly, I. Khoffi, L. Gasser, and B. Ford, "Enhancing bitcoin security and performance with strong consistency via collective signing," in *25th USENIX Security Symposium (USENIX Security 16)*, pp. 279–296, USENIX Association, 2016.

[76] E. Syta, I. Tamas, D. Visher, D. I. Wolinsky, P. Jovanovic, L. Gasser, N. Gailly, I. Khoffi, and B. Ford, "Keeping authorities" honest or bust" with decentralized witness cosigning," in *Security and Privacy (SP), 2016 IEEE Symposium on*, pp. 526–545, Ieee, 2016.

[77] Y. Minsky and A. Trachtenberg, "Practical set reconciliation," in *40th Annual Allerton Conference on Communication, Control, and Computing*, vol. 248, 2002.