KTH ROYAL INSTITUTE
OF TECHNOLOGY

Degree Project in Computer Science

Second cycle, 30 credits

Practical Analysis of the Giskard Consensus Protocol

LEON SANDNER

Stockholm, Sweden, 2023

Practical Analysis of the Giskard Consensus Protocol

LEON SANDNER

Master's Programme, Computer Science, 120 credits
Date: September 26, 2023

Supervisor: Karl Palmskog
Examiner: Mads Dam
School of Electrical Engineering and Computer Science
Swedish title: Praktisk analys av Giskard Consensus Protocol

Abstract

Consensus protocols are the core of modern blockchain systems, such as the Bitcoin, Ethereum, and Algorand networks. Thanks to these protocols, participants in a blockchain network can reach consensus on which blocks to add to a blockchain, to have a consistent chain of blocks in the whole network. Blocks are typically collections of transactions, e.g., transferring currency between participants, but can contain other data depending on the use case.

Consensus protocols are difficult to design, test, and implement. Problems may only manifest in rare cases but can have disastrous consequences. Formal protocol models allow exhaustive analysis and formal verification. However, formal verification is only as good as the specification of the protocol. For example, specification properties could be vacuously true, or the specification may omit to mention liveness properties that are crucial for progress in a system implementation.

This thesis describes the practical validation of a formal specification of Giskard, a consensus protocol used in the PlatON network. We translated key aspects of a formal model of Giskard in the Coq proof assistant to executable Python code, and integrated them into the Sawtooth blockchain framework. Then, we ran simulations of a blockchain network executing Giskard using Sawtooth, checking network node state properties of Giskard as well as its global safety properties. Automated clients from the Sawtooth network provided randomly generated transactions as input for the network. The implementation with the test results are available for reproducibility.

Both consensus safety and liveness issues were found in the formal specification of Giskard during the simulations. Based on the previous informal English specification of Giskard, we propose a new formal specification which was validated to reach consensus on blocks in the simulated blockchain network and uphold the protocol's crash and Byzantine failure tolerance to a certain degree. The improved formal model can serve as a new basis for verifying and implementing Giskard in the future.

Keywords

Distributed Ledger, Blockchain, Consensus Protocol, Giskard, Hyperledger Sawtooth

Sammanfattning

Konsensusprotokoll utgör kärnan i moderna blockkedjesystem som Bitcoin-, Ethereum- och Algorandnätverken. Tack vare dessa protokoll kan deltagare i blockkedjenätverk nå konsensus om vilka block som ska läggas till, för att få en konsekvent blockkedja i hela nätverket. Block är vanligen samlingar av transaktioner, till exempel överföringar av valuta mellan deltagare, men kan också innehålla annan data beroende på användningsområdet.

Konsensusprotokoll är svåra att designa, testa och implementera. Problem kanske bara visar sig i sällsynta fall, men kan ha katastrofala konsekvenser. Formella protokollmodeller tillåter uttömmande analys och formell verifiering. Men formell verifiering ger bara så goda resultat som protokollspecifikationen tillåter. Till exempel kan specfikationsegenskaper vara sanna innehållslöst eller så kan specifikationen underlåta att nämna framstegsegenskaper som är viktiga för att en systemimplementation inte ska låsa sig.

Denna avhandling beskriver den praktiska valideringen av en formell specifikation av Giskard, ett konsensusprotokoll som används i PlatON-nätverket. Vi översatte nyckelaspekter av en formell modell i bevisassistenten Coq till körbar Python-kod och integrerade koden i blockkedjeramverket Sawtooth. Sedan körde vi simuleringar av ett blockkedjenätverk som använder Giskard med hjälp av Sawtooth och undersökte Giskards tillståndsegenskaper för nätverksnoder och även dess globala säkerhetsegenskaper. Automatiserade klienter från Sawtooth-nätverket tillhandhöll slumpgenererade transaktioner som indata för nätverket. Implementeringen med testresultaten är tillgängliga för reproducerbarhet.

Både säkerhets- och framstegsproblem hittades i den formella specifikationen av Giskard under simuleringarna. Baserat på en tidigare informell specifikation av Giskard på engelska föreslår vi en ny formell specifikation som validerades att nå konsensus på block i det simulerade blockkedjenätverket och att till en viss grad tolerera krascher och byzantinska fel. Den förbättrade formella modellen kan användas som en ny bas för att verifiera och implementera Giskard i framtiden.

Nyckelord

Distribuerade Huvudbok, Blockchain, Konsensus, Giskard, Hyperledger Sawtooth

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

A fundamental problem in decentralized networks with faults is for network nodes to reach an agreement, or consensus, on values [1], or more practically, reach an agreement on a *ledger* of transactions between system clients [2]. The seminal work by Nakamoto [3] demonstrated that reaching an agreement on a distributed ledger of transactions in the presence of Byzantine, or adversarial, faults is practically possible under modest assumptions when it is computationally expensive to add blocks of transactions. Byzantine faults meaning, all possible kinds of faults that could appear. This approach gave rise to the Bitcoin cryptocurrency and its successors, but also motivated research into *alternative* consensus protocols, like Giskard, that do not require solving computational puzzles to add blocks [4]. Blocks consist of bundled transactions of several clients in a network. Blocks are linked through the cryptographic hash of the previous block, so all blocks are linked to their previous and next block, forming a chain [3], see Figure 2.1. This makes it possible to validate the chain of blocks from start to end, from parent block to child block, to create trust in the ledger for everyone.

Since the integrity of the distributed ledger is of high importance in networks with Byzantine faults, considerable effort is often spent on theoretical analysis of new and existing consensus protocols, in particular using formal methods such as interactive theorem proving [5, 6]. However, even when desirable properties of consensus protocols are mathematically proven and the proofs are machine-checked, the analyzed model of the protocol may be far removed from its actual implementation and deployment, potentially leading to unexpected behavior and bugs [7, 8]. This is why an executable implementation is needed, to practically analyze, and confirm if Giskard is a viable consensus protocol for blockchains.

Giskard is a Byzantine fault-tolerant three-phase consensus protocol in the partially synchronous mesh communication model, used to reach agreement on blocks added to the distributed ledger in the PlatON network. The implementation is based on only publicly available information, mostly the two specifications of Giskard.

- English specification by Li et al. [9]
- Formal model code written in the Coq proofing assistant [10]
- Report describing the model code [11]

All have to be seen as separate specifications in this thesis.

1.1 Motivation

The specification and model were never implemented or otherwise practically validated, the only known implementation is proprietary. The PlatON network contains the only practical implementation of Giskard, and the network displays that its consensus protocol works. There might be big gaps between the abstract Giskard specification and the implementation by PlatON, which is not publicly available. Due to those possible deviations, should Giskard be viewed as a separate protocol from PlatON's actual consensus protocol. Several useful safety properties of Giskard were formally verified [11, 10], but that still leaves the question if it works in practice as well.

This is why Giskard should be implemented within a simulated blockchain network to show that the proposed, abstract safety properties of the protocol hold, and if it can reach consensus on a chain of blocks, to show that is an alternative to other blockchain consensus protocols.

1.2 Problem and Research Questions

The objective of this thesis is to validate the formal Giskard model and specification of Li et al. [9, 11], by implementing the protocol within a modular blockchain framework, and simulating it in a realistic setting, while checking the required safety properties. For the reproducibility of results, will the implementation of Giskard be made publicly available as open-source software. As a secondary objective, should the implementation be reusable by developers of distributed ledgers, e.g., for practical comparisons with other consensus protocols on specific transaction workloads.

Research Questions:

1. Can the abstract Giskard consensus protocol as described by Li et al. [9] in mathematical English, and formally in the Coq proof assistant [11], be implemented to run in a real-world distributed ledger?

2. Can such an implementation achieve consensus in the presence of Byzantine faults for realistic workloads of proposed blocks of transactions, while guaranteeing the abstractly specified safety properties?

1.3 Scope and Limitations

Gaps between the formal model [11] and the specification by Li et al. [9] might exist, limiting the accuracy or feasibility of an implementation. Gaps could have been created by interpreting specific behavior of the protocol in a different than originally intended way. A bigger gap probably exists between the specifications and the implementation by the PlatON network [12], as the formal specification of Giskard was derived from PlatON, but the authors of the specifications got only limited information about the original consensus algorithm, creating this bigger gap. The implementation by PlatON is already working in practice, and we will only focus on publicly available information about it in this thesis.

It might also be that some properties of the formally proven model might not work in a simulated environment with transmission lag, message loss, or Byzantine behavior of nodes. This would not render the project useless but show that the protocol needs to be adapted and fixed. This might be done in this project, as long as there is enough time and there is nothing fundamentally wrong with the protocol.

What will also not be tried, is testing the behavior of the protocol that is not covered by the formal model code. Testing the specified behavior is already a lot of work, and without verified model code, can we not come to a definitive conclusion on if the executions in the implementation were actually according to the protocol.

Another limitation is the complexity of the simulated environment. We will not test the Giskard implementation in truly large-scale networks, but only in manually instrumented tests with a hand full of nodes.

The implementation is aimed to integrate into the Hyperledger Sawtooth modular blockchain. The auxiliary functionalities of the other Sawtooth consensus protocols will not be implemented, to limit the scope of this thesis.

Auxiliary functionalities like the IAS (International Accounting Standards) API, transaction families, or example transactions, like the tik-tak-to package.

Also out of scope is to make the repository an official Hyperledger project, as this requires much dialog and time-consuming revisions to be fully compliant with their requirements. This might be attempted after the thesis is completed as it is only of secondary importance.

1.4 Thesis Structure

Chapter 2 gives the necessary background knowledge for this thesis. It starts with explaining distributed ledger technology with a focus on blockchains, Section 2.1. Then, is a rough overview of consensus in distributed systems given, Section 2.2, which then goes into detail on properties of consensus algorithms, Section 2.2.1. Afterward, are foundational consensus protocols presented, and then specific blockchain consensus protocols with the Giskard protocol in much detail, Section 2.2.2. In the end, related work is discussed, Section 2.3, how companies and researchers are trying to build modular distributed ledger frameworks. To make it easier to swap certain parts of a blockchain to make it easier to test them, specifically the consensus protocol, or to just simulate a blockchain with specific, known network issues.

In Chapter 3, is the research methodology explained, what paradigms are followed, and what the research process looks like. The research process section, Section 3.2, shows what is planned to be done during this thesis. It contains a comparison of what distributed ledger framework is most suitable for easy modular consensus implementation, the simulation setup, test scenarios, and the reproducibility of the results. The ethics of the thesis are also shortly discussed in Section 3.3.

The actual implementation is described in Chapter 4. It describes how the implementation was transferred from the formal specifications [9, 11], the integration of the translated code into Hyperledger Sawtooth. The chapter shows in detail how the simulation and data collection were implemented, as well as what tests were done, and how they were validated. Furthermore, are the found and fixed bugs of the coq model code explained in detail, Section 4.4.

Chapter 5, shows the results and findings of the conducted tests, as well as a discussion about the implications and limitations of these results, Section 5.2. And finally, Chapter 6 concludes the thesis and gives an outlook into what future work can be done to improve on the implementation, and what more tests and comparisons should be done.

Chapter 2

Background

In this chapter is background knowledge provided to understand the topics of this thesis. Distributed ledgers, Section 2.1, more specifically blockchains, Section 2.1.2, and in great detail background knowledge on consensus protocols, Section 2.2.3, which the Giskard consensus protocol ultimately is.

2.1 Distributed Ledger Technology

With the help of a ledger can different parties achieve consensus on the things recorded on it. Typically, is ownership recorded, like assets in a company, or money in bank accounts, as well as transactions between accounts, to trace back the change in ownership, and flow of goods. The ledger needs a trusted party, which manages the ledger, records all transactions, and keeps the accounts. This is historically done by a trusted central party like a bank. For money transfers from bank to bank is a payment system needed on top of the ledger, like *Visa* [13]. But there are many privacy concerns regarding banks, credit cards, or online banking, because of data collection and reselling of customer data [14]. People also need to trust the central bank, which issues the currency of their country, and which should try to stabilize its value, by for example regulating the supply of money. As banks went bankrupt during the 2008 financial crisis [15], broke trust down in those central parties, and new ways were explored to manage ledgers in a trusted way without central parties. Protocols to ensure trust in a decentralized system existed already, like the *Practical Byzantine Fault Tolerant* protocol (PBFT), or *Paxos*, more in Section 2.2.2. But Paxos is used for state-machine-replication (SMR) and is only crash failure tolerant. SMR is used for failure redundancy if a server crashes, and another one can take over with the replicated state.

PBFT on the other hand, doesn't scale well with many nodes [16]. The first feasible solution proposed was *Bitcoin*, a decentralized P2P permissionless electronic cash system [3]. With it, all participants in the network help create trust, instead of a single central party, and it showed through time that it is a robust system tolerant to Byzantine failures [17]. It was made possible through advancements in P2P-decentralized systems, data structures, and cryptography. Later, other projects like *Ethereum* [18], or *Algorand* [19] adapted the Bitcoin network, for making more decentralized applications (DAPPs) possible than payments.

The Bitcoin system includes its own currency, the *cryptocurrency Bitcoin*, a payment system for making transactions, and wallets for users where Bitcoins or fractions of it are stored. It is a Distributed Ledger Technology (DLT), more specifically a blockchain. Not all distributed ledgers store transactions as a blockchain structure, but can for example be stored as a distributed hash table (DHT) [20] or *Directed Acyclic Graph* (DAG) [21, 22]. DLTs can be *permissionless* or *permissioned*. Permissionless networks like Bitcoin are also called public, as everyone can join the network by downloading client software and connecting to it. Permissioned DLTs, like *Hyperledger Fabric* [23], are private and mostly deployed by companies that want control of who can become a user, and what permissions a user has. Permissions about what resources a user is allowed to read, or what kind of transactions a user can make. As permissioned DLTs have some kind of central authority, could those systems also be implemented as other kinds of distributed transaction networks. This is if the involved parties trust the setup of authority in the network. Another drawback of DLTs is that they don't scale as well with a growing number of nodes, compared to centrally managed networks with trusted transaction managers. DAGs promise limitless scale, but this has to be yet achieved, as there is no true decentralized DAG network yet. Bitcoin handles around 7 transactions per second (tps) [24]. Proof-of-stake based blockchains tend to have higher tps, with Algorand claiming to have 6000 tps [25]. Existing distributed transaction frameworks are capable of handling more tps, like Apache Cassandra [26] with up to 1,000,000 tps. Visa states their network can handle about 24,000 transactions per second [27].

2.1.1 Participating in a Distributed Ledger network

So how does it look like participating in a DLT network. A DLT network's availability depends on the data structure of the DLT and on how fast it can process transactions. In blockchains is the system available after the local

node of a user has connected to several other nodes, and received the newest block from them, from which point it can make transactions or participate in committing new blocks of transactions to the network. Committing a transaction requires it to be collected in a block, which needs to be validated by other nodes, then appended to their local blockchain, and then the transaction sender needs to be updated with the newly added block with its transaction. Have enough other nodes received the committed block, is it said to be accepted by the network. For a more technical, in-depth overview, see Section 2.1.2.

In DHT networks like in Holochain, connects a client to the system, receiving a share of the system storage in the form of a partition of the DHT of the application it tries to run, like a messaging app [20]. It is then able to send an encrypted message to another participant, abiding by the rules of the application for making such a transaction. Other nodes also running the same application will then validate the message, by checking these rules. Is it valid, is it appended in their local log of messages and shared via a *gossip protocol* [28], where nodes exchange information with their nearest neighbors, which then pass on this information further throughout the network. Does a node validate wrong transactions, is the node flagged, other nodes are notified and block the node from further participation.

DLTs with DAGs as their data structure are not implemented yet without a central party for consensus, as they are a very recent development. DAGs are designed to be the fastest in network connection and transaction commitment, as they only need to receive and validate a few previous transactions to commit their own [21], which then needs to be verified by the next node wanting to commit.

All permissionless DLTs are decentralized without any central party, with some like Bitcoin allowing for heterogeneity, or different kinds of functionality, of the participating nodes in the network. So-called *light nodes* do not participate in committing new blocks but can make transactions. They need *full nodes*, which connect them to the network and gather their transactions. So the Bitcoin network can be seen as a hybrid network with some kind of central nodes, which are still decentralized as long as they do not belong to a single party. The light nodes have several full nodes connected, to prevent the forging of their transactions. Permissioned DLTs or semi-permissioned DLTs have some kind of third party, building trust in the network and controlling which node can or cannot participate.

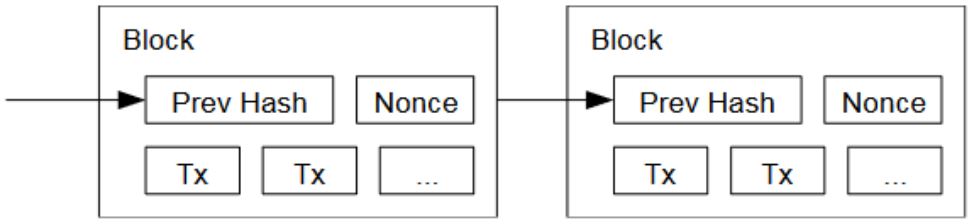

Figure 2.1: The structure of a blockchain. Blocks are linked, by storing the hash of the previous block, thus making the chain tamper-proof. The only way of altering a transaction in a block is to change the transaction, generate the hash of the block, and fill in the hash in the next block, which changes its hash, which changes the hash of the next block, and so on. As new blocks of transactions are coming in regularly, and other nodes in the network need to be convinced that the altered chain is the correct one, is it like a 'race'. This racing with the correct chain is only possible to win, if the attacker has more nodes in the network than the consensus protocol is capable of defending against, and if the attacker's nodes are faster at altering the past blocks up to the newest [3]; Image source: Nakamoto [3]

2.1.2 Blockchains

The blockchain is a derivative of the hash-chain structure [29], where a password is stored in the form of a chain of hashes of the original password. Does a user want to authenticate to a system, requests the system the hash of the previously sent hash, and so on. A blockchain, on the other hand, is about transactions of digital coins, that are listed in a block, which is stored in a chain of blocks. The blocks are linked, through the cryptographic hash of the previous block, so all blocks are linked to their previous and next block, forming a chain [3], see Figure 2.1.

The hash of a block is the hash of all block data, including the transactions, and the hash of the previous block. This makes it very compute intense to alter an established blockchain, as one would need to alter all blocks, from the block to change, to the highest, as the hash value of the altered block would differ as soon as its data has changed, which would invalidate all higher blocks, as they stored the original hash of it.

Block height stands for its position in the chain. The first block is called *genesis* block, and has the block height zero. Every new block added to the chain, increments this number by one.

Hashing is a one-way function of creating a so called *fingerprint* of data [30]. In other words, it transforms data into a specific sequence, which

only this specific data could have generated. Like fingerprints identifying a specific human. It is called a one-way function, as it is easy to calculate the hash of some data, but very hard to turn the hash back into the original data, when the exact hash function is not known [30], this demo gives a hands-on explanation of how hashing is used in blockchains [31].

A node of a user makes an upcoming transaction public, by sending it to nodes that it is connected to [3]. Has a node collected enough transactions, do blockchains usually create a Merkle tree [32], with the hashes of the transactions as leave nodes, the branch nodes as hashes of the childs' hashes, and the root as the hash of all hashes in the tree. This way, when a transaction is altered, will the hashes of the branches and the root hash change, which is easily detectable. Depending on the consensus protocol is it proposing the new block to other nodes in the system. This is done, by sending the block to other nodes which test the transactions, by verifying the chain of ownership of the sent coins, and the hashes within the Merkle tree. This works, as all nodes have the complete chain with the Merkle roots, and can look up all past transactions, and verify if the sender actually owns the sent tokens, see Figure 2.2. This also separates the confirmation of transactions and the actual transaction data. Through linking a block with a branch of the Merkle tree where the transaction was in, and seeing that a node previously accepted the transaction, as its hash was stored in this branch [3].

Is the block valid, do the nodes store it in their local storage. Other nodes can see if their transaction was successful when enough connected nodes have stored the block, containing the transaction. This is a rough overview, every DLT implements the validating and gathering of transactions in their own way. This is done via a consensus protocol, building trust between the parties involved. Section 2.2.2 gives general consensus protocol examples and specific examples from DLTs.

2.2 Consensus in Distributed Systems

Lamport et al. [33] famously illustrated the problem of reaching consensus with the *Byzantine Generals Problem*. It describes the scenario of a group of generals of an army, encircling an enemy city, and the generals need to reach consensus on what time to attack together. The generals only communicate via messengers, which may be unreliable, or deliver messages of traitor generals, who try to confuse the others. The generals have to cooperate and come to an agreement, so a consensus protocol needs to be found to solve this problem.

In computer science, reaching consensus is important for safety critical

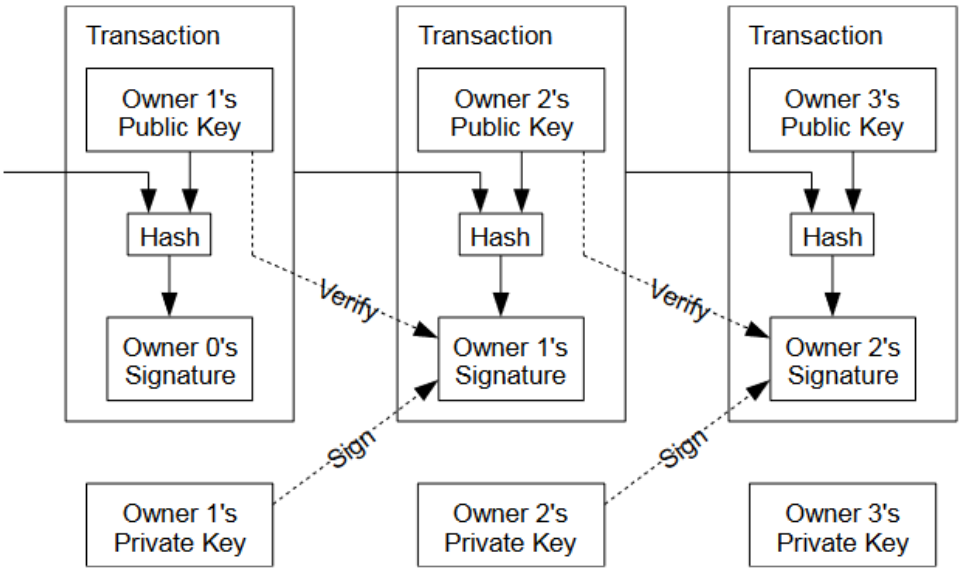

Figure 2.2: Linking of token transactions, for validating that a user actually owns the spent token or coin. Trailing back a token along the chain of its transactions is done, by verifying the previous owner's signature via its public cryptographic key. The signature itself is created with the previous owner's private key. The transactions are linked via the hash of the previous transaction, plus the public key of the next owner of the token; Image source: Nakamoto [3]

systems, like spacecrafts [34], later airplanes [35], medical devices, and now even cars with autonomous driving capabilities [36]. Featuring redundant hardware components that need to agree on sensed and computed values. Sensors that sense the environment, and computers perform operations on the sensed data. They need to reach consensus so every device has a consistent view, or state, with the same data. For the case that a sub-system fails or has inaccurate results, and the system is still operating on valid information. Otherwise, contradicting sensor values, of for example two accelerometers in a rocket, could lead to wrong flight path calculations, and in the worst case a self-triggered explosion, not to hit residential areas when crashing.

The other prevalent domain where distributed devices need to reach consensus is transaction-based systems [37], like distributed databases, banking systems with distributed ledgers, and blockchains. There, the focus lies on the consistency of transactions, stored on distributed servers of a database, bank, or nodes of a blockchain network. Consistency is important here, as inconsistent data between the servers could mean that on one server,

a customer of a bank has 0€, and on another one 1000€. A transaction has not been received on one server, or the transaction was not possible in the first place, as the sender did not have enough funds, and one server didn't check. So the servers need to reach consensus on what transactions happened, in what order, and if a transaction was successful or not [37].

The mentioned systems are all of distributed nature. The redundant components of the safety critical systems, the banking servers, and the participating nodes of a distributed ledger network are spatially distributed, interconnected, and exchange information via message exchange, like in the Byzantine Generals Problem. Message transmissions over unreliable networks like the internet, have variable latency, message loss, and devices can fail or act maliciously, so inconsistencies between servers can arise regularly, so countermeasures have to be taken to avoid inconsistencies. How each system solves consensus, depends on its architecture and choice of consensus protocols, which will be described in the next sections.

2.2.1 Consensus Protocol Properties and Requirements

In this section will consensus protocol properties and requirements be explained. Consensus in shared-memory systems will not be discussed, as it is not important for this thesis, but is an interesting area still. For more information, is this a good starting point [38, Ch. 4, 5, 6].

Coulouris et al. describe a basic state model for a distributed system trying to reach consensus [39, Ch. 15.5]. The system consists of processes or nodes $p_i \; i \; \varepsilon \; \mathbb{N}$, they start out in the *undecided* state and will propose a value for a vote. The origin of the value depends on the protocol, the processes could have generated it themselves, or have received it from a *leading* process, which decides on what to vote on in a certain round of elections. The processes communicate with each other and exchange their values, and then decide on a value, named *decision value* $d_i \; i \; \varepsilon \; \mathbb{N}$, thus reaching the *decided* state. The value proposing here stands for the last round of value proposing, it can follow rounds of preceding value exchanges. Coulouris et al. propose three requirements a consensus algorithm has to fulfill during all its executions, *termination*, *agreement* and *integrity*.

- Termination, means that eventually all correct processes will set a decision value, so they individually will decide on a value.
- Agreement, requires the decision value d_i of all correct processes to be

the same $d_i = d_j \; i \neq j$.

- The integrity property demands all correct processes to propose the same value, and if they do, every correct process will choose this value in the decided state.

What is meant here with correct process, is a process and message exchange free of failures, the next section goes more into detail about different kinds of failures and failure detection, which is an essential part of consensus algorithms.

2.2.1.1 Failures while Reaching Consensus

An important property of a consensus protocol is its resilience to failures. Also called *t-resilience* [40], where t stands for the maximum number of faulty processes for which the protocol can still reach consensus. This thesis uses f for faulty instead. Is there one more faulty process, can the protocol not guarantee anymore that the agreed-upon value is the one with a majority of votes. As the typical way to reach consensus on a value, is to vote on it. Is a certain majority of votes reached for the same value, is it used as the result of the vote. Some protocols also use the minimum or maximum value [39, Ch. 15.5], as well as random values, when a required majority couldn't be found.

Faulty behavior is usually separated into two categories, *crash failures* and *Byzantine failures*. And a consensus protocol has an *f-resilience* for crash failures, and or Byzantine failures. A crash failure stands for a process stopping all its activities [40]. It is the simpler case of failures, whereas Byzantine failures include for example, the process coming back alive with an old state, or the process only sleeping for some time and then sending an answer, message latency, message loss, or other messaging errors. More difficult-to-handle Byzantine failures include malicious behavior, where processes are aiming to disrupt the voting process in any way. Sometimes a process can also have no malicious intend and display disruptive behavior when there is an undiscovered bug [40]. Byzantine failures include all possible kinds of failures, also crash failures.

2.2.1.2 Failure Detection

In some cases, it might be important for a consensus protocol to know if a process failed [40], especially in the case a leading process failed. A leader, or primary, is often responsible for initiating and managing votes on values or being the single proposer of a value within the current round of voting.

Detecting failures can be very difficult though, as there are endless possible Byzantine failures that would need to be specifically handled. Most often, timeouts are being used to detect crash failures. For that, participating nodes need to be synchronized in time and need to have time bounds, in which they should respond to a message or send so-called *heartbeat* messages [41], showing that they are alive. Lamport et al. provide an overview of time synchronization [42]. Lamport et al. also provide an algorithm that detects when a process passes on wrong values, or 'lies' about its own [43]. Their proposed algorithm uses so called *authenticators*, a supplement of data, authenticating that the sent values are indeed from the original sender. Common encryption algorithms can be used for this [44]. Processes have a private and public key-pair, sign a message with their private key and thus generating the *authenticator* data, provided with the to be sent message. With the help of the signature and the public key of the sender, can the receiver verify that the message was from the sender, and has not been tinkered with by another process forwarding the message. The only problem there is to distribute the public keys between the processes reliably and securely, an RSA asymmetric public key exchange was proposed to solve this problem [45].

2.2.1.3 Upper and Lower Bounds in Consensus protocols

There are lower bounds on how resilient different kinds of consensus algorithms can be. The theoretical lower bound of asynchronous consensus protocols to be non-resistant to even one crashed process was shown [1], so an f-resilience of 0. Synchronous, or semi-asynchronous protocols have a theoretical lower bound of $f + 1$, where f is the amount of faulty processes and $f + 1$, the amount of communication rounds the protocol needs to reach consensus. The upper bound for Byzantine failures was also proven [40], which is $f < \frac{1}{3}p$ where p is the number of processes. So no protocol can guarantee to reach consensus, when there are equal or more than $\frac{1}{3}$ processes faulty.

2.2.1.4 Asynchronous & Synchronous Protocols

Another property of consensus protocols is how communication is handled, fully asynchronous, synchronous or semi-asynchronous. Asynchronous messaging means, no upper bound on how long a message can be expected to be received, like in a system with remote participating processes. Fischer showed that in such a case, no process can be faulty for consensus to be reached [1]. In transaction-based systems, like in banking, when a transaction

is sent to several transaction managers, that need to execute the transaction on their local state, to verify that it is valid, all transaction managers need to agree on the transaction [1]. This is because every transaction manager needs to have a consistent history of transactions. When a new transaction comes in from a client, it is forwarded to all transaction managers that are responsible for it. A fully asynchronous protocol allows that a transaction manager's reply to commit a new transaction has no timeout, so no failure detection. If it will never reply, the system will wait forever, and won't make any progress [1], failing the previously mentioned termination property by Coulouris et al. [39, Ch. 15.5]. Committing a transaction means deciding on it and storing it in a local database.

To circumvent this issue of non-termination, are failure detectors applied, using time-outs or heartbeat messages. This makes an asynchronous system, semi-asynchronous though, a mix between fully asynchronous and synchronous, with relaxed time-bounds after which they will be seen as crashed.

Synchronous systems consist of highly synchronized distributed processes. They work in so-called *lock step* or rounds [40], a strict time-bound in which they have to act, like proposing a value. Here failure detection is automatic, as in one step or round, a message is received, processed and the reply is to be sent in the next round. This requires highly synchronized clocks though, which requires its own synchronization protocol with a lot of extra message exchanges [42].

Semi- and fully-synchronized systems can guarantee some form of fault-tolerance [40], but are more complex the more synchronized they become, which has to be considered when designing a consensus protocol.

2.2.2 Foundational Consensus Protocols

One of the earliest formulated consensus protocols is *Paxos* [46], it servers as a distributed SMR protocol. It is designed to be crash-tolerant, but only if crashed nodes come back alive, reload their state successfully and eventually send responses. It is an asynchronous protocol meaning that it does not have any timeouts to detect failing nodes, so it is actually non-failure tolerant, due to the impossibility theorem, see Section 2.2.1.3 and Section 2.2.1.4. It does not guarantee termination or *liveness* but only that when a value is chosen, can the other nodes only choose this value. Figure 2.3, shows the architecture of Paxos. The protocol has two phases, preparing and proposing. A proposer sends a new value x with a proposal number n to its connected acceptors,

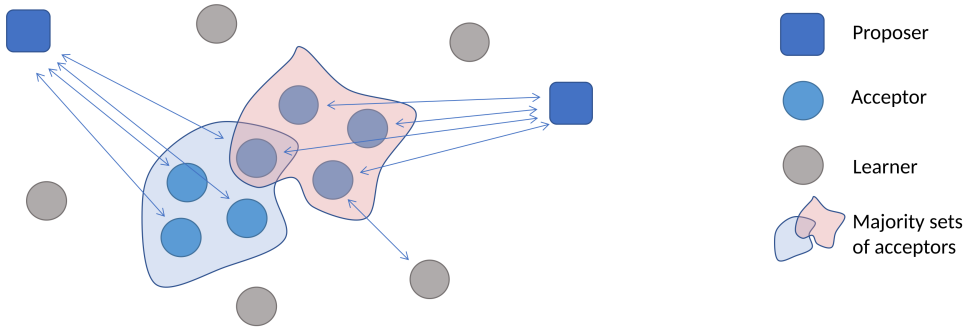

Figure 2.3: Paxos consensus protocol architecture. Its nodes are separated into *proposers*, *acceptors*, and *learners*. Proposers receive new client transactions and try to connect to a majority of acceptor nodes, to propose them a new value. Acceptors only accept the newest value they have received, so if there are several proposers, does only the fastest get its value accepted by the necessary majority of nodes. Learners can learn the newest accepted value from the acceptors [47].

$prepare(x, n)$. The acceptors respond with their last received value with the highest local proposal number $acc(y, m)$. If the proposer receives a majority of accepts, sends the proposer a $propose(y, n)$ message, where y is the value of the highest received *acc* message, or its own value x if no acceptor had a value yet. If a majority of acceptors acknowledge this $propose(y, n)$ message, is the value y chosen and the acceptors store it with the proposal number in their local memory. Does an acceptor have received a value with a higher proposal value in between the *prepare* and *propose* messages, does it not send an *ack* message back to the proposer, which leads to the proposer not having enough votes to chose the value, and needs to try again in another round of the protocol.

Paxos has recovery procedures for recovering acceptors, learners and proposers. Does a learner crash, is it not important to the consensus process, as learners just read the newest value. Does an acceptor fail during a vote, does it only matter if the proposer does not have a majority set anymore, and should try to connect to more nodes. Does a proposer fail during a vote, is it detected via timeouts or randomization [47], and a new one is elected. Later, the *Raft* consensus protocol adapted Paxos, to be easier to understand by engineers [41].

While Paxos and Raft are crash tolerant, albeit not guaranteed, is the PBFT protocol, Byzantine failure tolerant up to $f < \frac{n}{3}$ [48]. It is a semi-asynchronous protocol, as its messages are bounded by timeouts, which

makes it possible to detect crash failures or lost messages. Messages are authenticated to secure against Byzantine replay or spoofing attacks. Its main purpose is also SMR. Its nodes are separated into primaries and backups for failure redundancy, yet other implementations of PBFT make multiple leaders possible. It requires three rounds of communication, a *Pre-Prepare*, *Prepare*, and *Commit* phase, making leader election in more cases of failure possible [48]. Unlike Paxos, does it guarantee termination and liveness, as well as agreement and integrity.

2.2.3 Consensus Protocols in Blockchains

To create trust within a blockchain network while proposing new blocks, is it typical that a node needs to provide proof that they have a stake in the network, to reduce the likelihood of the node wanting the network to fail. This can be done in several ways, the original blockchain network Bitcoin, as well as Ethereum's old IstanbulPBFT protocol, used Proof-of-Work (PoW), while more recent protocols use Proof-of-Stake (PoS) [49], Proof-of-Elapsed-Time (PoET) [50], Proof-of-Capacity (PoC)[51] or Proof-of-Activity (PoA) [51].

PoW, or *Nakamoto style lottery*, is implemented in Bitcoin via a search for a *nonce*, a random value with which the block's hash will lead with a certain amount of zero bits. This act of searching for a hash of the block with leading zero bits is called *mining*. Is a block mined by a node, does it get a service fee reward and an amount of bitcoin which is newly created or *minted*. The rate at which a new block is mined in the network should stay stable, to keep the rate of issuing tokens the same, for bitcoin tokens to not lose value too fast through an oversupply of tokens. The more miners try to find the hash for the next block, the more likely it is that the next hash is found sooner, as more processors will try out random nonces, to find the right combination. Every 2,016 blocks is the amount of needed leading zeros adjusted, more leading zeros if blocks were mined faster than before. This is more difficult to find the hash for, as the probability of finding consecutive zero bits declines per required zero. The rate at which connected nodes try to find the next hash per second is called *hashrate* [52]. The hashrate of a blockchain network based on PoW, determines its security from attacks by attackers with 51% of nodes in the system, as explained in Figure 2.1. This is because the higher the hashrate, the more processing power the attackers need to amass to arbitrarily change past transactions. The logic behind this is, that if a party has a high stake in the network, through for example processors connected to it, does it also want the network to function and stay alive. The amount of processing done in PoW

has negative effects as well, being the massive amount of energy used, when the network scales up and has more miners competing for a reward [52].

PoS on the other hand uses only the monetary stake in the system, to create trust and dependency on the network[49]. Here, not the first node to mine a block is chosen to propose a new block, but nodes are randomly selected depending on the amount of tokens they have staked. Staking is the process of reserving an amount of tokens for a certain amount of time, to participate in proposing new blocks. Ethereum migrated from PoW to PoS [49], because it also allows for potentially faster transaction speeds and less energy consumption, as no random nonce needs to be generated for a hash to have leading zeros.

2.2.4 Giskard Blockchain Consensus Protocol

Giskard, the blockchain consensus protocol that is being implemented in this thesis is a derivative of PoS [9]. It is Byzantine failure tolerant when less than a third of nodes are Byzantine. It achieves this for example, by letting each message be signed and subsequent messages have aggregated signatures, to prevent spoofing messages, similar to the authenticators, mentioned in Section 2.2.1.2. It is a semi-asynchronous protocol, with bounded timeouts. Some literature uses the term *safety* instead of the properties integrity and agreement, and *liveness* instead of termination. A common explanation for liveness is, that something good will eventually happen, whereas with safety, that something bad will never happen. For a more in-depth explanation see Alpern et.al [53]. The formal specification of Giskard has three safety properties [9] relating to integrity and agreement. The first is that no two blocks of the same height can reach local prepare stage in the same view. The second is two blocks of the same height that are in local pre-commit stage, have to be the same blocks. The third is two blocks of the same height that are in global commit stage, have to be the same blocks. Its safety properties are formally verified, but its liveness property is not, which the implementation in this thesis will test.

The protocol separates distinct spans called *epochs* e_n for $n \in \mathbb{N}$ which contain further stages separated into *views* v_n $n \epsilon \mathbb{N}$, in which blocks are being proposed. In an epoch e_i, a set of k participating nodes $N_i \subset \mathcal{N}$ is randomly selected, depending on their amount of staked tokens. Then, pose the selected nodes N_i as proposers and validators. There is one proposer, and the rest act as validators. This distinction of nodes is similar to Paxos' proposers, acceptors, and learners, here the learners are nodes that were not selected for the epoch.

After the proposer proposed $10 \cdot k$ blocks, and when they have reached global prepare stage, ends the view v_i, and another validator poses as a proposer, starting the next view v_{i+1}. The nodes change in a round-robin fashion to be the proposer, until 250 blocks have reached global prepare stage, ending the epoch.

Blocks have different stages locally on a node, and globally across all participating nodes. The *prepare stage*, when a block b_n of height n is initially proposed, the *precommit stage*, when a block's child block has reached prepare stage, and *commit stage*, when the child block has reached the pre-commit stage through a vote quorum, confirming that a majority validated the block. To increase throughput, can the proposer prepare the 10 blocks in parallel, not needing to wait for a vote quorum one by one. Thus, the production and validation of blocks are separated, like in the *HotStuff* protocol [54]. The messages send for the blocks to reach the prepare stage is $PrepareBlock(b_i, n, v)$ with i, the block index from 0 to 10 in the current view, n the sender's node index in the epoch, and v the current view number. This message is sent to every validator, and itself. Figure 2.4, shows the structure of a participating node, and the flow of messages.

Has a node a $PrepareQC(b_j, v_,)$ in its counting message buffer for the parent of the block, or does it have a vote count $count_b \geq N - f$, or does the prepare message already contain a PrepareQC of the parent block, is the prepare message stored in $l_{counting}$, otherwise sends the node a $PrepareVote(b_i, j, v)$ to the other nodes. Does a node receive a $PrepareVote(b_i, i, v)$, does it check if it already send a PrepareVote, did it not, does it send a PrepareVote for it and checks if it already received enough votes for the block to reach a vote quorum. Did it reach a quorum, and does it have a $PrepareVote(b_{i+1}, i, v)$ message of the child of the block, does it send a $PrepareQC(b_i, j, v)$ for the block to the other nodes, which commits the block.

This *normal behavior* is when there are no timeouts of the proposer, or a majority of nodes, called normal behavior. The protocol's messages are bounded by timeouts, which are calculated for each view by an algorithm known by all nodes. With the timeouts and counting of blocks per view, can nodes check for a failing proposer or failing of a majority of validators, and issues a ViewChange. Is a ViewChange quorum reached between the alive nodes, is the view changed via ViewChangeQC messages. The highest block in local prepare stage is sent with each ViewChange message by the participating nodes. The highest block received in this message exchange is used as *carryover* block, the parent block of the first block in the next view.

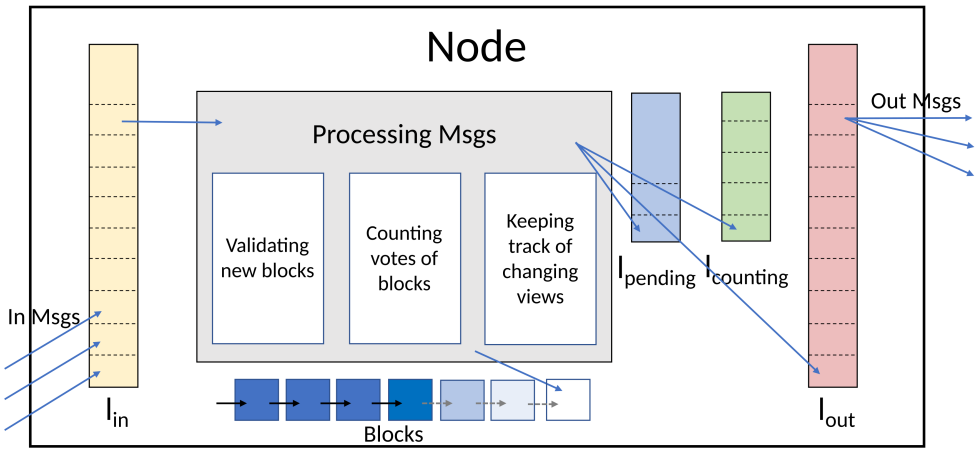

Figure 2.4: Structure of a Giskard node, with the flow of messages. A proposer processes received messages like it would be a validator. Messages are received and stored in an *input message buffer* l_{in}, they are then processed and stored in a *counting message buffer* $l_{counting}$. Has a node received the $PrepareBlock(b_n, i, v)$ message does it validate its transactions, invalid blocks are discarded. $l_{pending}$ is for received PrepareVote messages, where the corresponding PrepareBlock message hasn't been received yet. Those votes are checked as soon as the block has been proposed to the node. The node keeps track of the number of blocks, to know when to change to the next view. The l_{out} message buffer is for messages that should be sent to other nodes, like PrepareVote, PrepareQC, ViewChange and ViewChangeQC. Blocks are locally stored. Blocks in dark blue are blocks from a previous view and are globally committed. The transparent block is in local prepare stage. The parent block has reached a vote quorum and as it has a child block in local prepare stage, is it in local pre-commit stage. The grandparent block in a darker shade is in local commit stage, as it also reached a vote quorum and has a child block in pre-commit stage.

For even more details on the protocol, see the formal specification [9], or the formal verification [11, 10].

2.2.5 Giskard Model in the Coq Proof Assistant

The formal model of Giskard in the Coq proof assistant [10] takes the common approach of defining a transition system step relation between *global states* containing individual node states and in-transit network messages. Each permitted step in the transition system then corresponds to some individual action by a node or other network event.

To define node actions, the Coq model axiomatizes a small theory of blocks. Some of the operations on blocks, such as block height, are only partially specified, i.e., only specified in sufficient detail to allow formal proof of safety properties. Based on the theory of blocks, the model then defines network messages and the state of individual nodes, as shown in Listing 2.1.

```
Parameter block : Type.
Parameter GenesisBlock : block.
Parameter b_height : block → nat.
Parameter b_index : block → nat.
(* ... *)
Parameter generate_new_block : block → block.
(* ... *)
Parameter parent_of : block → block.
(* ... *)
Record NState := mkNState {
 node_view : nat;
 node_id : node;
 in_messages : list message;
 counting_messages : list message;
 out_messages : list message;
 timeout : bool;
}.
Definition NState_init (n : node) : NState :=
  mkNState 0 n [] [] []  false.
```

Listing 2.1: Coq model code: axiomatization of blocks and definition of node state (structures.v)[10]

The Giskard step relation in Coq is encoded as a predicate on pairs of global states, where a global state is defined as a tuple consisting of a function from node identifiers to states and a list of in-transit messages. A step is either a *node step* or a *timeout step*. A node step performs some node-local action that may process a message, change node state, and send new messages, as defined by a node transition relation. Listing 2.2 shows a fragment of the Coq code defining the Giskard step relation (`GState_transition`). Note that the specific node transition relation used in a node action is obtained by passing a node transition type (`NState_transition_type`) to the function `get_transition`.

```
Inductive NState_transition_type :=
| propose_block_init_type
  (* ... other node transition types ... *)
```

```
| process_PrepareBlock_malicious_vote_type.
(* ... *)
Definition get_transition (t : NState_transition_type) :
 (NState → message → NState → list message → Prop) :=
  match t with
  | propose_block_init_type ⇒ propose_block_init
    (* ... other node transition relations ... *)
  | process_PrepareBlock_malicious_vote_type ⇒
     process_PrepareBlock_malicious_vote
  end.
(* ... *)
Definition GState : Type := (node → NState) * list message.
(* ... *)
Definition GState_transition (g g' : GState) : Prop :=
  (∃ (n : node), (* node action *)
    In n participants ∧
    ∃ (process : NState_transition_type)
     (msg : message) (lm : list message),
       get_transition process) (fst g n) msg (fst g' n) lm ∧
     g' = broadcast_messages g (fst g n) (fst g' n) lm)
     ∨ (* timeout *)
     g' = ((λ n ⇒ if is_member n participants
             then flip_timeout (fst g n)
             else fst g n), snd g).
```

Listing 2.2: Coq model code: global states and transition relation.

Protocol traces in the Coq model of Giskard are abstractly encoded as functions from natural numbers to global states, where each adjacent state (e.g., the global states for 2 and 3) are related by the step relation. Listing 2.3 shows a fragment of the relevant Coq code. This definition allows properties of reachable global states (from the initial state associated with 0) to be proven by induction on natural numbers.

```
Definition GState_init : GState := (λ n ⇒ NState_init n, []).
Definition GTrace : Type := nat → GState.
(* ... *)
Definition protocol_trace (t : GTrace) : Prop :=
  t 0 = GState_init ∧ ∀ n, GState_transition (t n) (t (S n)).
```

Listing 2.3: Coq model code: protocol traces.

2.2.6 Coq Refinement to Executable Node Functions

While the transition system encoding described in Section 2.2.5 is convenient for formal proofs of safety properties, such a definition entangles node actions and network actions [55]. That is, the node actions need to be (manually) separated from the definition of global transitions to implement a protocol for nodes following the model. This task is made easier by the Coq formalization defining node state transition relations modularly, e.g., having separate relational definitions for different messages, as shown for the PrepareBlock message type in Listing 2.4. However, the node transition relations only indirectly describe the result of node actions in terms of state updates and messages to send.

```
Definition process_PrepareBlock_vote (s:NState) (msg:message)
 (s': NState) (lm:list message) : Prop :=
  s' = (* Record outgoing PrepareVote messages *)
   record_plural (process s msg) (pending_PrepareVote s msg) ∧
  lm = (pending_PrepareVote s msg) ∧
  received s msg ∧
  honest_node (node_id s) ∧
  get_message_type msg = PrepareBlock ∧
  view_valid s msg ∧
  (* PrepareBlocks cannot be processed during timeout *)
  timeout s = false ∧
  prepare_stage s (parent_of (get_block msg)).
```

Listing 2.4: Coq model code: definition of a node transition relation

To address this problem, the second author of the Coq formalization [10] refined node state relations to executable Coq functions, i.e., to function definitions in Coq corresponding to each specific node transition relation used in `get_transition` in Listing 2.2. As an example of refinement to functions, consider the original relational definition of processing of Giskard PrepareBlock votes in Listing 2.4 and the corresponding function definition in Listing 2.5. Note that the function `process_PrepareBlock_vote_set` uses *record update* notation to express how an input state is transformed to an output state by setting new record field values. However, properties that do not express state transformation or the outputted list of messages to send are omitted and must be checked elsewhere, such the absence of a timeout.

```
Definition process_PrepareBlock_vote_set (s : NState)
(msg : message) : NState * list message :=
```

```
let lm := pending_PrepareVote s msg in
let s' := s
 <| in_messages := remove message_eq_dec msg s.(in_messages) |>
 <| counting_messages := msg :: s.(counting_messages) |>
 <| out_messages := app lm s.(out_messages) |>
in (s', lm).
```

Listing 2.5: Coq refinement code: definition of a node transition function

2.3 Related Work in Simulating Blockchain Networks

The most important related work is the specification and formal verification of the Giskard protocol [9, 11]. They are the basis of this thesis. Section 2.2.2, provides related work on similar protocols, like the foundational Paxos consensus protocol, and the more related Bitcoin PoW or Ethereum2.0 PoS protocols. Proving certain aspects of blockchains is a relatively new field of study. Several projects try to formulate and implement modular frameworks for simulating blockchains for research, already reaching different specific goals [56, 57, 58]. Some blockchain frameworks are more focused on simulating network behavior through varying transfer latencies and other message transfer behavior that may arise while having nodes connected over the internet [59, 60, 61]. This is also of interest, as Giskard should be tested against various crashes or Byzantine failures. However, these frameworks are more focused on specific blockchain networks, like Bitcoin or Ethereum.

For this thesis was a framework used that was as modular as possible and not specialized for a certain blockchain. Hyperledger Sawtooth was used instead of JABS or Scorex, mostly for the fear of not having enough documentation or no help from the original creators during the implementation process, and for the wish of getting the implementation one day to a level where it can be used in a live test network. A comparison of those frameworks and blockchains is provided in Section 3.2.1.

Chapter 3

Methodology

3.1 Research Method

The research method is based on (1) software *runtime verification* [62] and (2) software *model-based testing* [63].

More specifically, we implement the Giskard protocol, closely according to the formal model [10], in a distributed ledger framework. We analyze the implementation by generating executions. These executions generate traces of actions, or logs, made by each network node and messages passing through the network. The traces are the input to tests that check the safety properties (runtime verification), which are also derived from the formal model. In the tests, we compare the system behavior to an idealized trace (model-based testing), according to transition properties from the model code. Transitions are the nodes possible valid transitions, according to the formal model. The resulting list of property violations and deviations from an idealized trace are then analyzed, manually and using visualization tools such as message diagram charts, and tables, showing details on the proposed and voted-on blocks. The implementation is adapted until it is equal to the idealized trace, or the formal model contains bugs that hinder liveness in the protocol execution, which should be fixed. Simulations are manually instrumented, to test general behavior and specific edge cases of Giskard. Transactions are randomly generated, which are bundled up into blocks and passed to the consensus protocol as input.

3.2 Research Process

The research process consists of the following steps:

1. Manual translation of node behavior specified in the formal model to executable code.
2. Checking that the implementation was faithfully translated from the formal model.
3. Analysis of the differences between the implementation and the model.
4. Runtime checking of consensus reachability.
5. Measuring quantitatively, to validate the tests.

The simulation-based analysis of the implementation of Giskard tries to establish that the protocol has actually been implemented as specified, i.e., that there is a correspondence between simulation steps and the abstract global steps in the formal model. In mathematical terms, the ideal is to prove the existence of a *refinement mapping* [64] between the global transition system and the transition system given by the implementation. In practical terms, evidence for a mapping can be provided by generating executions and checking that changes to node state are consistent with possible changes in the model to abstract global states. Whether a concrete execution refines an abstract transition is a qualitative measure.

The simulations try to establish whether the formally specified safety properties for Giskard actually hold at all times during executions. Given an execution trace, this should be a yes or no question, and therefore a qualitative measure. If there is a violation of a property, the manual analysis should try to establish whether, for example, the implementation or the abstract model is at fault. If the model, and by extension the protocol, is at fault, can this potentially have scientific value to the distributed systems research community and give insights into hurdles with the practical implementation of consensus in blockchain systems.

The execution logs are being analyzed, to detect when or if consensus is established among nodes on what transactions to add to the distributed ledger. This is again a qualitative measure.

Quantitative measures are taken, for validating the executed tests. The measurements consist of the number of recorded states, and tested state transitions, to evaluate if every transition was actually tested after the simulation.

3.2.1 Choosing a Blockchain Framework

The blockchain framework Hyperledger Sawtooth was chosen for implementing Giskard, and simulating executions of the protocol. Using an existing blockchain framework reduces the effort compared to developing everything from scratch. A framework should be able to provide the generation of transactions and bundling them up into blocks, as well as providing the software for nodes that participate in the network and communicate with each other, via the exchange of messages.

Name	Modularity	Maturity	Documentation	Scalability
Hyperledger Fabric [23]	✗	✓	✓	✓
Hyperledger Sawtooth [50]	✓	OK	✓	✓
Ethereum [18] & Bitcoin [3]	✗	✓	✓	✓
Substrate [65]	OK	✓	✓	✓
Hyperledger Scorex [57]	✓	OK	✗	OK
BlockSim [59]	OK	OK	OK	OK
VIBES [61]	✗	OK	✗	OK
JABS [56] & TOOL [58]	✓	OK	✗	OK

Table 3.1: Comparison of blockchain frameworks, for consensus protocol implementation and simulation. Light gray colored columns are frameworks specifically made for simulating blockchains.

Table 3.1, shows an overview of the comparison of considered blockchain frameworks, for implementing Giskard. The main difference between the listed is that some are fully developed and deployed blockchains, like Bitcoin, or Hyperledger Fabric, and some frameworks, marked in grey, are made specifically for simulating and testing certain aspects of blockchains. Having a framework that is made specifically for testing, would be an obvious choice, but several other factors are important too. The most important for making the implementation as straightforward as possible, is the modularity of the framework [66], especially considering the modularity of the consensus layer. Modular systems have only a few connections between the modules, making

changes to a module less complex, as only a few other things in the system would need to be changed as well. A modular consensus layer would make it possible to implement Giskard in one module and swap it with the existing consensus protocol module. Most deployed blockchains are not developed with modularity in mind. They were developed for a specific purpose and usually don't fundamentally change, like Bitcoin. Ethereum on the other hand had a quite major change, by changing to PoS consensus [67]. The frameworks marked ✗ in the "Modularity" column, have their consensus layer too spread out over their system, making it too cumbersome to change it to Giskard. Some simulation frameworks like VIBES or BlockSim were also implemented with some specific consensus algorithms in mind, so they are also not suitable for this thesis. Hyperledger Sawtooth, Scorex, JABS and TOOL, have the ability to swap the consensus algorithm, making them the most suitable for this thesis. Taking the other listed properties into consideration, is Hyperledger Sawtooth the most suitable framework. It is compared to the other modular frameworks, better documented, and has a bigger developer community, which helps in understanding the system and for getting help when problems arise. In hindsight could the JABS framework have been a better fit, as it is a smaller framework which could have made setting it up, and implementing Giskard easier.

3.2.2 Simulation Setup

Figure 3.1, shows the setup for the blockchain network simulations. The Sawtooth framework handles the creation of blocks, and provides the software for validator nodes and clients, which automatically connect to each other and exchange messages. Clients generate batches of transactions during the simulations, which are sent to all nodes. There, they are bundled up into blocks and sent to the Consensus Engines for validation. The consensus engines run the Giskard consensus protocol. On receiving, and processing a message, update the Consensus Engines their local state, and send their state to the Giskard Tester, similar to I/O automata [70]. I/O automata are models for nodes in or components of a distributed system. An I/O automaton is a state machine, reacting to events, which lead to local state transitions, which can be recorded and tested. The Giskard Tester keeps a global state transition trace of the network, for later state transition comparison to the model transitions, and safety property tests. Chapter 4 shows how this was implemented in detail, and Chapter 5 shows the results of the orchestrated tests.

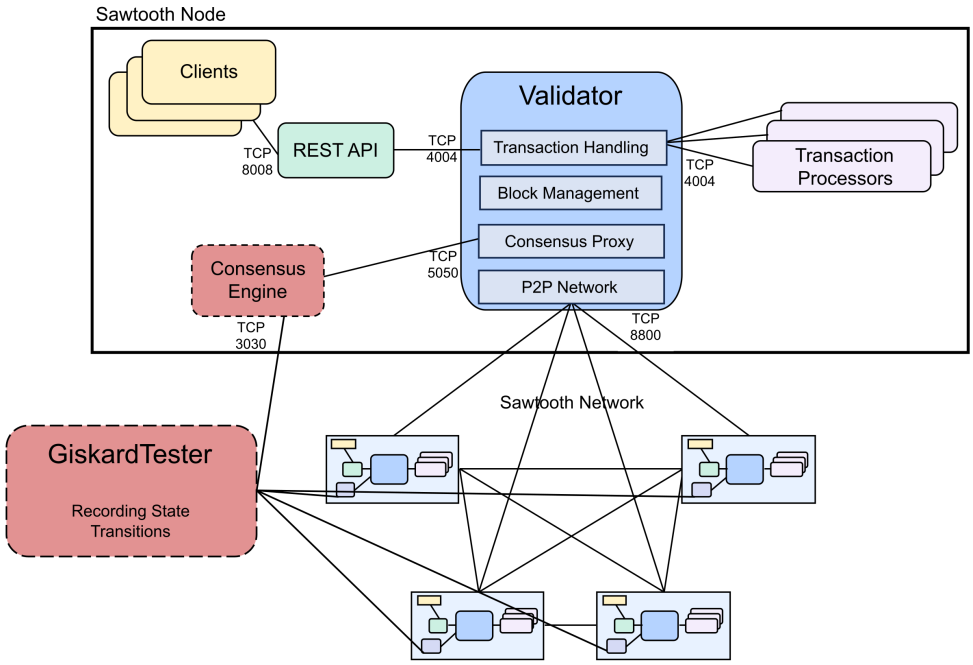

Figure 3.1: Simulation setup for recording the tests. A Giskard Tester is connecting to the *Consensus Engines* of all Sawtooth nodes in a Sawtooth network. It is creating a global trace of state transitions, by receiving and recording local state updates via ZMQ TCP sockets [68]. Both, marked in red, were written for this implementation. The consensus engine code was adapted from existing Sawtooth code, from the PoET consensus protocol; Image adapted from Sawtooth [69].

3.2.3 Test Scenarios

The test scenarios should confirm normal behavior of the protocol and cover at best, all edge cases of the original specification [9], that are also covered by the model code. See Table 3.2 for the chosen test scenarios, and for the test results, see Chapter 5. More test scenarios could be imagined, like testing more kinds of possible malicious behavior, but no model code exists to verify those executions of the implementation. This thesis limits itself to only what can be verified by the formal model.

3.2.4 Reproducibility

The tests should be easily reproducible and the test results readily available. The implementation is archived on Zenodo [71]. It replaces the code from

Category	Test Scenario
Normal Behavior	Normal view changes due to three blocks reaching global prepare stage per view, not through timeouts. Not starting any malicious nodes, so no proposal of different same height blocks, nor voting for them.
Abnormal Behavior	Timeout after no block was proposed in the current view
	Timeout after first block of the view reached global prepare stage
	Timeout after second block reached local prepare stage in the node that times out
	Timeout after second block reached global prepare stage
	Timeout after last block reached global prepare stage
	One malicious node out of four double voting
	Three malicious nodes out of four double voting

Table 3.2: Test scenarios covered in this thesis. Normal behavior is described in Section 2.2.4, proposing new blocks, voting, reaching quorums and non-timeout view changes. Edge cases covered by the model code are different cases of abnormal view changes due to timeouts, and double voting of malicious nodes.

the PoET consensus protocol of the Sawtooth framework. The Sawtooth framework is written in Rust and Python, and requires packages that require Ubuntu 18.04 (Bionic) [72] and support Python up to version 3.6.9. Setting up the network with the PoET protocol poses already much effort, so the additional effort by switching to the Giskard implementation and running the tests should be as simple as possible. Instructions for the setup and running of the tests can be followed in the main README file in the Giskard repository [71]. The test results of this thesis can be found in the folder ”tests/sawtooth_poet_tests/test_logs/” [71], an overview of the folder structure is given in Figure 4.1.

3.3 Ethics and Sustainability

The implementation is publicly available [71], so it could be used in any project. Even in cryptocurrencies, or nft-projects, as those can utilize the implementation as their consensus algorithm. Those are known to be a place for money laundering and scams.
The major sustainability problem with such projects is the massive electricity use of distributed ledger projects if they use Proof-of-Work (PoW) as their consensus mechanism, like Bitcoin. The Giskard protocol uses Proof-of-Stake (PoS) instead, which is much less compute-intensive than PoW, and thus needs far less electricity. Even though, distributed ledger projects in general use a lot of electricity for upkeep, depending on the amount of nodes and transactions. If the protocol proves more efficient, the Jevons paradox [73] shows that energy consumption in total will not go down, but up. At least more transactions might be able to be processed with the same amount of energy than before.

Chapter 4

Implementation

We describe the implementation process in great detail in this chapter. To serve as an entry point for understanding the implemented code. Furthermore, to get a better grasp of what has been done, why, and what the difficulties were. First, did we translate the formal model code to Python, Section 4.1, which we then integrated into the Hyperledger Sawtooth blockchain framework, Section 4.2. Then, we test the implementation by recording a global trace of local state transitions of the participating nodes, while the network is running to reach consensus on blocks, Section 4.3. While comparing the recorded global trace with the ideal transitions provided by the model, did we find bugs in the formal model code, which we could fix, Section 4.4.

4.1 Translating Coq Model Code to Python

After setting up a local installation of the Sawtooth network, was the next step to replace parts of one of the existing consensus protocols with the behavior of Giskard. An existing protocol was replaced instead of writing everything from scratch, as a lot of code is required for connecting the Consensus Engine, where the consensus protocol is situated, and the validator. There are two consensus protocols implemented for Sawtooth, PoET, written in Python, and PBFT written in Rust. Both could have posed as skeletons for implementing Giskard. PoET was chosen, as it seemed easier to set up and alter for this implementation, as well as being written in Python, where the style of the functional features in the language seemed more similar to the Coq code than Rust, making translating the code easier. Translating the model code to Python by hand was done, instead of using automatic transpilation from Coq to Haskell or OCaml, as the chosen framework uses Python. Also, a trial

transpilation showed that not everything was successfully transferred, and the generated code was not easy to read. One could have used the transpiled code and put it in a different blockchain framework, but the variety of blockchain frameworks written in Haskell or OCaml is very limited. We could also have transpiled the code further to Python, but due to the unusability of the extracted code was that not tried. Direct transpilation from Coq to Python does not yet exist to our knowledge. The model code also leaves many executable parts of Giskard open, as the Coq proving assistant does not need them for its proofs, so a big part needs to be implemented by hand anyways.

As described in Section 2.2.5, does the Coq model define data structures and operations for messages, blocks, states, nodes, global states, and global traces with varying degrees of detail. Translating these data structures and operations to Python cannot be done line-by-line, in particular, due to the axiomatization of blocks and block operations shown in Listing 2.1. The axiomatization omits a lot of information that is typically needed for a concrete implementation, like the payload of a block, the signer, and the parent block id. The parent relation is modeled by introducing the parameter *parent_of* in line 8, which leaves this parameter open for interpretation. This is in contrast to the definition of *NState*, Listing 2.1, where the data structure is given as a record of the necessary parameters, which leaves less room for interpretation of the code.

In addition to data structures, are also functions needed to work with them. The model code provides typical functions, definitions that require certain inputs, and returns a new output, like in Listing 4.1. Here, a state and a message are required as input, the output is a new state with the message removed from the in buffer, and added to the counting buffer. The Python version is also given as an example for a straightforward translation, see Listing 4.2.

```
1 Definition process (s : NState) (msg : message) : NState :=
2  mkNState (node_view s) (node_id s)
3   (remove message_eq_dec msg (in_messages s))
4     (msg :: counting_messages s) (out_messages s) (timeout s).
```

Listing 4.1: Coq model code: definition of the function *process*, as an example for a simple function (local.v)[10]

```
1 @staticmethod
2 def process(state: NState, msg: GiskardMessage) -> NState:
3     state_prime = Giskard.discard(state, msg)
4     state_prime.counting_messages.append(msg)
```

```
5     return state_prime
```

Listing 4.2: Python implementation code: definition of the function *process*, as an example for a translation from the Coq model code (giskard.py)[71]

Most interesting are the state transition definitions, which take a state, and the next state after a transition, as input to define a valid transition. This is then used to prove that a global transition system, where the transitions must fulfill the specified properties, only leads to the specified valid transitions. For example the definition *process_PrepareBlock_vote*, Listing 4.3, and its translation to Python, Listing 4.4, it models the behavior of a node receiving a *PrepareBlock* message, where the parent block has reached prepare stage, so the node votes for the proposed block in the message. These functions are more complex, but are straightforward to translate, as they leave little to no room for interpretation, unlike some functions that they use, like *generate_new_block*, Listing 2.1.

```
1  Definition process_PrepareBlock_vote (s:NState) (msg:message)
2   (s': NState) (lm:list message) : Prop :=
3    s' = (* Record outgoing PrepareVote messages *)
4        record_plural
5          (process s msg)
6          (pending_PrepareVote s msg) ∧
7    lm = (pending_PrepareVote s msg) ∧
8    received s msg ∧
9    honest_node (node_id s) ∧
10   get_message_type msg = PrepareBlock ∧
11   view_valid s msg ∧
12   (* PrepareBlocks cannot be processed during timeout *)
13   timeout s = false ∧
14   prepare_stage s (parent_of (get_block msg)).
```

Listing 4.3: Coq model code: definition of the relation *process_PrepareBlock_vote*, as an example for a valid state transition property(local.v)[10]

```
1  @staticmethod
2  def process_PrepareBlock_vote(state: NState, msg:
       GiskardMessage, state_prime: NState, lm:
       List[GiskardMessage], node, block_cache, peers) -> bool:
3      parent_block =
       block_cache.block_store.get_parent_block(msg.block)
4      lm_prime = Giskard.pending_PrepareVote(state, msg,
       block_cache)
```

```
    return state_prime == \
        Giskard.record_plural(Giskard.process(state, msg),
    lm_prime) \
        and lm == lm_prime \
        and Giskard.received(state, msg) \
        and Giskard.honest_node(node) \
        and msg.message_type ==
    GiskardMessage.CONSENSUS_GISKARD_PREPARE_BLOCK \
        and Giskard.view_valid(state, msg) \
        and not state.timeout \
        and Giskard.prepare_stage(state, parent_block, peers)
```

Listing 4.4: Python implementation code: definition of the function *process_PrepareBlock_vote*, as an example for a translation of a valid state transition function, before its bugs were fixed(giskard.py)[71]

Definitions with the same importance as the transition properties are the height injectivity statements, see Listing 4.5 for an example. These represent the protocol's three safety properties, prepare stage, precommit, and commit stage safety. The tests iterate over a global trace of node transitions and fail, if there are two blocks of the same height in prepare stage in the same view, in pre-commit stage, or in commit stage.

```
Definition prepare_stage_same
_view_height_injective_statement :=
  ∀ (tr : GTrace), protocol_trace tr →
    ∀ (i : nat) (n m : node) (b1 b2 : block) (p : nat),
      In n participants →
      In m participants →
      b1 ≠ b2 →
      prepare_stage_in_view (fst (tr i) n) p b1 →
      prepare_stage_in_view (fst (tr i) m) p b2 →
      b_height b1 = b_height b2 →
      False.
```

Listing 4.5: Coq model code: definition of the safety property statement *prepare_stage_height_injectivity* (prepare.v)[10]

Not all definitions were translated, as not all were necessary for the transition, and safety property tests, which was the main goal of the thesis. The model code contains several lemmas, theorems and proofs. These were also only translated if necessary, and most served as unit tests for the definitions and properties they try to prove, it is out of scope for this thesis to translate them all.

4.2 Integration into the Sawtooth Hyperledger Framework

Thanks to the modularity of the framework can the consensus protocol be swapped, and entirely new ones written. To make the implementation as straightforward as possible, was the PoET consensus protocols repurposed. It has its own repository as its own little Hyperledger project, where all code was copied from, instead of creating a fork, as Giskard is its own protocol instead of a rework of PoET. The code serves as the Consensus Engine of the framework, see Figure 3.1, or Figure 4.2. To implement Giskard, had the model code not only be translated but had also to work with the framework. The modularity of the framework made it possible to keep the code in a functional style, without needing to think about the API of the framework, between the Consensus Engine and the rest of the framework.

For an overview of the folder structure of the mentioned parts of the implementation, see the file-tree 4.1. All functions were written in one separate class as static immutable functions [71, giskard.py], meaning they don't mutate the input variables, and produce the same output given the same input. This even makes it reusable in another blockchain framework, if someone would like to reuse this code. There are more languages for automatic transpilation from Python compared to Coq. Data structures got their own classes [71, folder: engine/sawtooth_poet_engine/] and left out parameters by the model code, that needed to be filled in. The place where the local state is kept and the functions of Giskard are called, is the *engine.py* file [71, folder engine/]. It contains the main loop of the Consensus Engine, which receives new blocks from the network, and broadcast messages from other nodes. Depending on the type of a received message and the local state, are different functions called, in Giskard's case, the corresponding state transitions. Here is also the connection to the Giskard Tester established. After receiving a message, and after a state transition, is the updated state sent to the Giskard Tester. The main difficulty here was to exactly replicate the protocol behavior, which was to be inferred from the transition properties. Most transitions model the behavior of the protocol receiving a specific type of message. For example, the PrepareVote message should only result in either the *process_PrepareVote_vote*, or *process_PrepareVote_wait* transition, or a timeout. So on receiving a PrepareVote message, must be the local state be analyzed, as well as the stored blocks, to whether the former or the later transition should be made, to exactly replicate the model behavior.

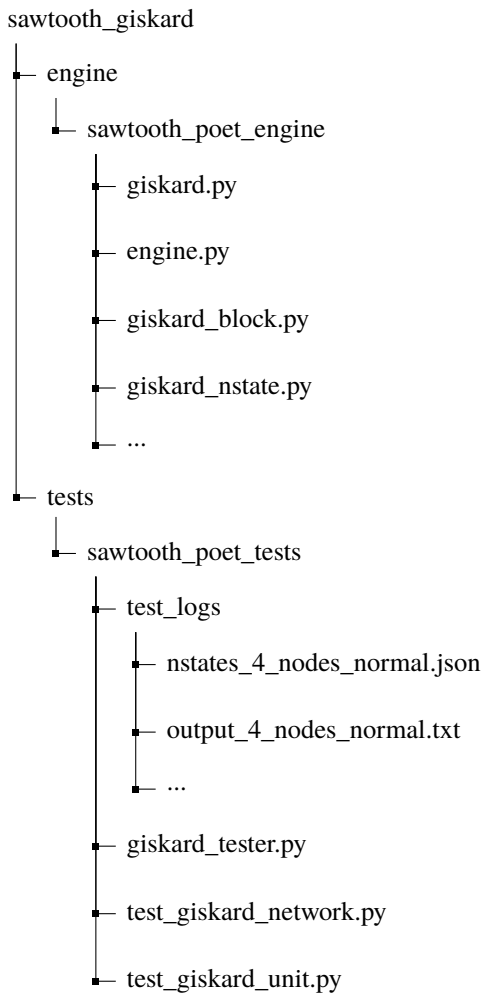

Figure 4.1: Folder structure of the implementation, only showing the important parts [71]. The translated model code is situated in the file giskard.py, and the data structures each in their own file in the same folder *sawtooth_poet_engine*. The integration, or network test is situated in the folder *sawtooth_poet_tests*, with the already produced logs in the *test_logs* folder.

4.2.1 Difficulties during implementation

For this, an exact understanding of the model code was necessary, though as Section 4.4 shows, made liveness bugs in the model code it impossible to get much progress done in the system. So the English specification [9] had to be taken into account, to fix the transition properties, and underlying functions, with the help of extensive logging and debugging of the network during an integration test. An integration test is a test where all parts of a system get

tested together, in contrast to a unit test, where only one specific small part of a whole system gets tested. An integration test tries to confirm that all parts work together as a whole, in other words, to validate system-wide behavior, in this case, the behavior of the blockchain network.

What also made the implementation difficult is that the model code often assumes that a specific messages or block just exist, and has no function to actually retrieve the message or block from the state. Often a quorum message and parent block is needed, but there are several possible places where this could be stored. Like in Listing 4.8, where a quorum message of a parent block is needed, for the function *pending_PrepareVote*. This quorum message could reside in the local counting-, or out-message buffer, or in the aggregated signature of the received message, when the parent block is the piggyback block of the message. Is the parent block the GenesisBlock, for which no quorum message exists, does an ad_hoc message need to be generated. For brefivity are those cases ommited in the Listing, with the function call to *get_quorum_msg_for_block*.

Several more issues had to be overcome to get to a working protocol and to reach consensus on blocks to add to the blockchain. At first was tried to write a standalone Consensus Engine, only reusing the skeleton code of PoET. For this, the rest of the framework would have needed to be appended information about the new protocol, and other changes to be made. The idea was first tried, but then abandoned, due to the amount of needed changes in the whole framework, which involves several repositories. The changes would have been small, but plenty, across all repositories. Adding the Giskard protocol as an option to start a network with, or change to, as well as starting a Giskard Consensus Engine, instead of a PoET or PBFT one from the command line, as a systemd service of Ubuntu, or as a docker instance. In most instances, only a name change from PoET to Giskard would have been necessary, as well as auxiliary information for the Giskard Engine, like if an honest or dishonest node should be started.

Another important issue is that the model code requires three blocks to be generated for a valid transition for a node to be the next proposer. The framework as of the writing of this thesis, only sends one block at a time to the Consensus Engine for validation, and after it has been validated and committed to the chain, sends the next one. Making it impossible for the proposer to propose three blocks at the same time. It cannot propose them one after another, as in the meantime, it would send and receive more messages, making the transition *process_PrepareQC_last_block_new_proposer* fail. The Sawtooth validator could be rewritten to send more blocks to the engine,

regardless of if they already reached consensus or not, but it is out of scope of this thesis. Instead, are the original functions of the PoET protocol still running in the background, receiving and committing blocks, to the network, and Giskard is running at the same time without interfering. So the framework is not yet using Giskard to reach consensus, but Giskard is running independently, only using the framework for receiving new blocks, starting, and connecting with all running nodes, which were the original requirements of the blockchain framework. This makes the implementation not fully usable for an actual blockchain, but the goal of this thesis is the validation of the consensus protocol and does not need to provide a fully functioning blockchain.

4.3 Testing and Recording State Transitions

Figure 4.2 gives an overview of the final simulation setup.
At first, were unit tests, derived from lemmas from the model code, used to confirm basic functionality and correctness of some of the translated code. However, as there were too many functions to write unit tests for, as well as many functions need complex inputs, which need to be generated, was early decided to write an integration test. An existing integration test was used to start a new network of a specified amount of clients that generate batches of transactions, as well as nodes with their respective Consensus Engine [71, test_giskard_network.py]. Starting a new network every time the test is run, means that there are no blocks yet in the network, a genesis block needs to be generated, with settings about the network, and started nodes have empty local blockchains as well.

4.3.1 Local State Recording Approach, Giskard Tester

At first, were logs of the Consensus Engines used, to track progress in the protocol manually, but this is not sufficient to prove the to-be-tested transition and safety properties. So an extra node called Giskard Tester was implemented, which connects to every Consensus Engine, and records all broadcasted messages as well as local state transitions. The recorded data is appended to a global trace of state transitions and broadcasted messages, which then serve as input for the state transition and safety property tests. An integration test is started with a specified amount of nodes, rounds, and batches to be produced by the clients. In a round, are batches produced and sent in different kinds of ways from the clients, so with the amount of rounds and

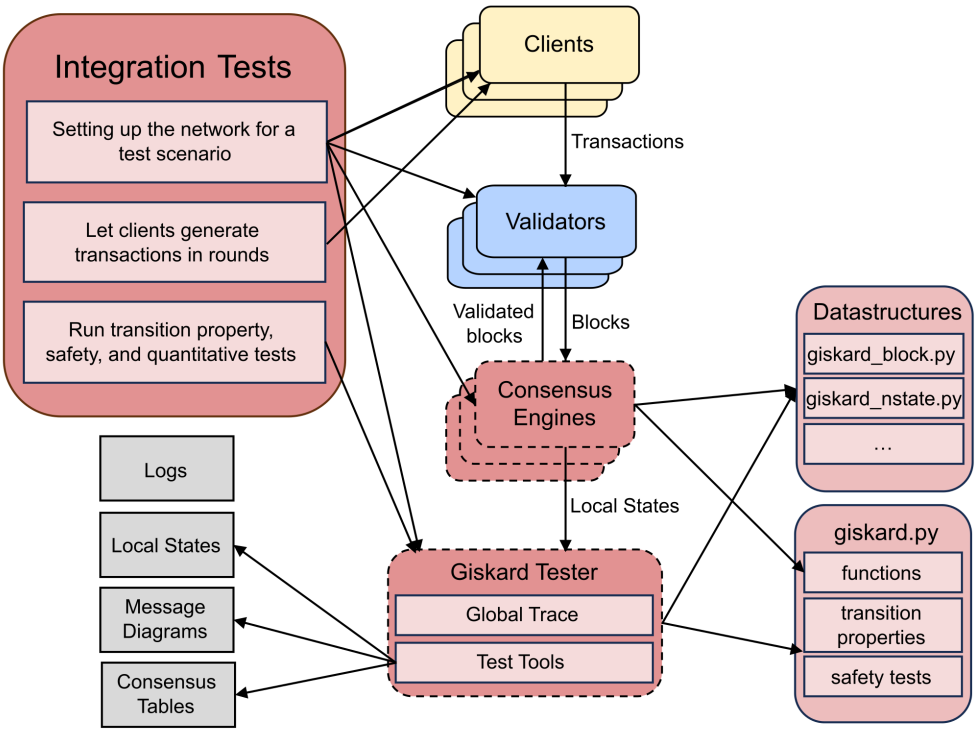

Figure 4.2: Architecture of the integration, or network tests. The integration tests test the safety and liveness properties of Giskard within a simulated blockchain network. Running one of the integration tests starts a new network according to a specified test scenario, see Table 3.2. The test then lets random transactions be generated which are bundled to blocks by the validators, which send them for confirmation to the consensus engines, where the nodes try to reach consensus on the blocks with the help of Giskard. During this, the local states of all consensus engine nodes are recorded, to form a global trace of state transitions. After several rounds of generating transactions, is the network stopped, and the safety and liveness tests are run, on the global trace of state transitions. After the tests are run, are the results stored in various kinds of formats for visualization, and reproducibility. Modules colored in red tones were written or adapted for this implementation.

batches, can the amount of blocks be roughly controlled. With the amount of nodes, can a more complex network be started, but for the carried-out tests were only two to five nodes used. Sometimes, even only five nodes overwhelmed the Giskard Tester, due to the amount of de-serializable data, which resulted in the process being killed by Ubuntu's Out of Memory Killer.

All nodes peer with each other, so the network's topology is a full mesh,

with $\frac{n(n-1)}{2}$ as the amount of connections needed, where n is the number of nodes. This results in exponential growth in the number of needed connections and with it, the amount of broadcast messages grows exponentially too. Two messages are needed per block from each node, broadcasted to every other node, a PrepareVote and PrepareQC message, as well as the initial proposal message of a block, from the proposer to all other nodes and to itself. Which amounts to $(2 + 2) \cdot \frac{n(n-1)}{2} + n$ messages. Is a valid message received, is a state update sent to the Giskard Tester. The node processes the message, and depending on the local state and the type of the received message, transitions the local state, where the resulting local state is also sent to the Giskard Tester, further increasing the amount of messages in the network. Not only does the amount of messages in the network grow exponentially but also the size of the local states sent to the tester. This is, as the full local state is sent, Listing 2.1, with all processed and sent messages, in the counting and out buffers. This is needed for the transition property and safety tests, which could be redesigned in the future to remove messages from old views, or only send the change in state instead of the full state. However, a normal deployment of the system would not include the Giskard Tester, and thus not this overhead.

4.3.2 Test Scenario implementation

To test specific timeout behavior or double voting by malicious nodes, can the test inputs be changed. There are four different kinds of timeout tests, and the amount of malicious nodes can also be set in the integration test *test_giskard_network* [71, test_giskard_network.py]. The timeout tests were chosen according to the edge cases specified in the Giskard specification, see Figures 4.3, and 4.4. The test scenarios differ in the height of the block in the view, that reached global prepare stage. The most interesting case is, case two in Figure 4.3, where node B is the proposer of the view, a timeout happens, and block b_1^B is the carryover block, the highest prepare block that reached global prepare stage. Block b_2^B reached local prepare stage in node B though, right before B had a timeout. So there can exist two same-height blocks locally. This is why the first safety property *prepare stage same view height injectivity* is more relaxed than the other two, pre-commit and commit stage height injectivity. It allows two same-height blocks to exist in prepare stage, but not in precommit or commit stage, as that would mean, that two same-height blocks reached global prepare stage, meaning they got included in the blockchain. Yet allows the prepare stage safety property only two same height blocks in different views, which happens in this case, as after a timeout,

an abnormal view change happens, leading to the next view.

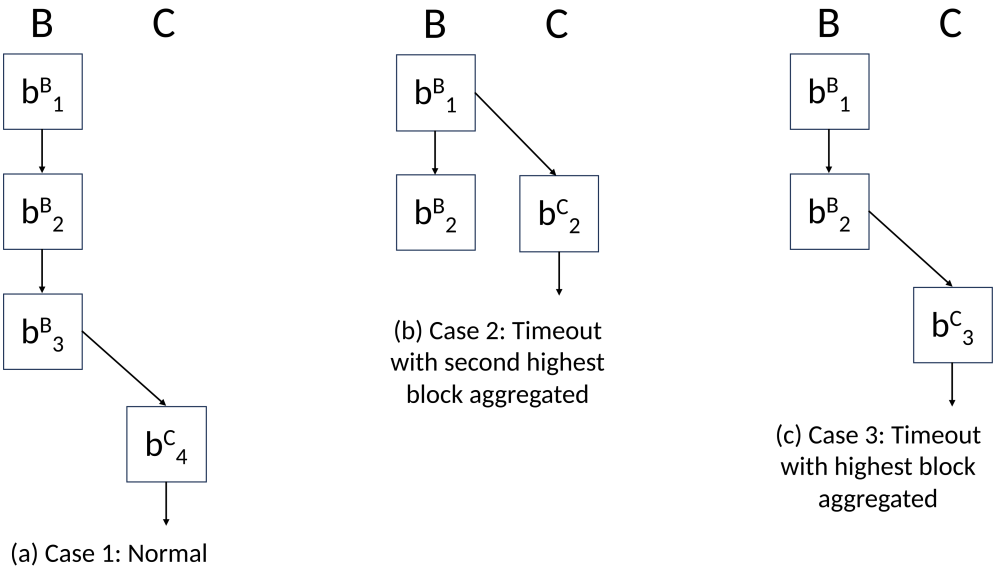

Figure 4.3: Three view change cases; Image adapted from Li et al. [9]

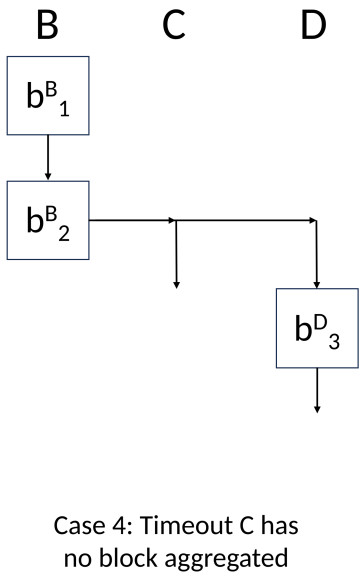

Figure 4.4: View change case with no Prepare stage blocks in the view of block proposer C; Image adapted from Li et al. [9]

Testing double voting is done by starting the network where a specified

amount of nodes are set to be dishonest, which at one point will introduce a malicious block, that does not stem from the correct proposer of a view. Honest nodes will discard these PrepareBlock messages, due to that reason. Malicious nodes will process this message and vote for this block, as well as the block of the same height, proposed by the correct proposer of this view. Depending on if the dishonest nodes can reach a vote quorum on the malicious block of over $2/3$, reaches the dishonest block global prepare stage or not, in addition to the correct block. The two-thirds majority is defined by Giskard as the necessary majority to reach a quorum.
Chapter 5, shows the results of the executed tests.

4.3.3 Implementation of the Transition and Safety Property Tests

The implementation of the transition and safety property tests posed more difficult than expected, as both tests require more than just the state before and after a transition. Both need the stored blocks, and the transition tests need the received message, the broadcast messages, if the node is dishonest or not, as well as the whole block cache. The block cache stores not only the committed blocks, but also pending blocks for future proposals, as well as blocks that reached a quorum during the current view. As testing with only sending the local states already resulted in the Giskard Tester being out of memory, are those inputs generated ad-hoc with only the information provided by the transitions, as well as the to be performed transition.

The transition property test goes through the whole recorded global trace, and tests pairs of consecutive states of each node, from start state to last recorded state. As the local states of the nodes don't include which transition they were the result of, does the test have to go through all possible transitions, each time generating the previously mentioned inputs, inferred from the to be tested transition and the two provided states, state and state prime. The global transition test succeeds, if all recorded transitions match the model transitions. It fails if no model transition is found that would result in state prime, if state prime is not the result of a timeout, where state prime equals state, but the timeout flag is set to true, see Listing 4.6.

```
1 Definition GState_transition (g g' : GState) : Prop :=
2 (∃ (n : node),
3   In n participants ∧
4   ∃ (process : NState_transition_type)
5     (msg : message)
```

```
    (lm: list message),
    (get_transition process) (fst g n) msg (fst g' n) lm ∧
    g' = broadcast_messages g (fst g n) (fst g' n) lm) ∨
g' = ((λ n ⇒ if is_member n participants
         then flip_timeout (fst g n)
         else fst g n), snd g).
```

Listing 4.6: Coq model code: definition of a valid global state transition, used to verify a global trace (global.v)[10]

Transitions were often not equal to the model transitions, as not only existed liveness bugs in the transitions but also due to the test setup. Is one change done in the implementation, might this need to be accounted for in three different locations. The Consensus Engine, needs to analyze the state in the correct way to choose the right transition, the ad-hoc generation of inferred inputs, and the transition property functions.

The safety property tests were already provided by the model code, for an example see Listing 4.5. The difficulty here was also the generation of the right inputs, which luckily could also be inferred from the pre- and post-states. The inputs needed, are the recorded states, the participating nodes, and the local blocks of each node. Each test iterates over all global state transitions, the cross product of all nodes, and the cross product of all local blocks two nodes have at the point in time of the global transition step. The prepare stage test is special, as it iterates over the cross-product of the local blocks and the views the nodes went through up to this global transition step, see Listing 4.5. For the reason that this test checks if two different same-height blocks are in prepare stage in the same view, and allows prepare stage of different same-height blocks in different views.

4.4 Fixing Liveness Bugs from the Model Code to Reach Consensus

Running the various integration tests, showed that many bugs exist in the model code, hindering progress in the network. 15 bugs in the model code have been identified and fixed until the protocol showed wanted properties and behavior in the tests. Ten various kinds of bugs in the transition property definitions, one extra transition that was needed, one bug in a function used by many of the transitions, two bugs that are not vital for liveness but cause unnecessary additional messages to be sent, and one typo in the transition mapping for the global transition test, which when solved, causes the safety

proofs in the Coq proof assistant to fail. Discussing all discovered bugs would be too much for this thesis, so only a selection of the most interesting is discussed. If there is interest, all bugfixes are marked in the implementation code, in the file *giskard.py*, with a comment beginning with *CHANGE from the original specification* [71]. All bug fixes are in accordance to the English specification of Giskard [9], which describes the protocol in text form and can be used for guidance on how to implement parts of Giskard not covered by the formal model code [10]. The bugs in the model code were mostly mistakes in which specific message or block was needed in which transition. What could be more easily solved, once one had a running implementation, where one can see what the actual input or state is of each transition.

4.4.1 Safety Bug in the Mapping of Transition Types to their Properties

The typo bug in the model code lets a transition property not being checked during the formal Coq proofs. Listing 4.7 shows the model code with the bug in lines 11-12, the mapping should be to the transition property *process_PrepareBlock_vote* instead. The bug ignores the transition where a node casts a vote for a block, after receiving a PrepareBlock message. This lets the transition property tests in the implementation fail, as no suitable model transition can be found for the recorded one. After the typo had been fixed in the model code, did the proofs fail, which couldn't be solved yet. On the other hand, the issue in the transition property test of the implementation was resolved. The issue is that no matching model transition could be found for the recorded transition during the integration test. The bugfix is according to the English specification, as it mentions that voting for a block right after receiving a PrepareBlock message is possible when one out of three conditions is true [9, Section 9.1, PrepareBlock]. Namely, three kinds of possibilities that the parent block has already reached a quorum, which is reflected in the *process_PrepareBlock_vote* transition.

```
1 Definition get_transition
2    (t : NState_transition_type):
3    (NState → message → NState → list message → Prop):=
4  match t with
5  | propose_block_init_type ⇒ propose_block_init
6  | discard_view_invalid_type ⇒ discard_view_invalid
7  | process_PrepareBlock_duplicate_type ⇒
8       process_PrepareBlock_duplicate
```

```
| process_PrepareBlock_pending_vote_type ⇒
    process_PrepareBlock_pending_vote
| process_PrepareBlock_vote_type =>
process_PrepareBlock_pending_vote
| process_PrepareBlock_vote_type =>
process_PrepareBlock_vote
...
end.
```

Listing 4.7: Bug in Coq model code: mapping of transition types to the respective transition property, for testing if all global transitions are valid. The bug highlighted in red, with the fix in green (local.v)[10]

4.4.2 Liveness Bugs in the Transition Properties

The ten bugs in the transition property definitions were found across eight different transitions, so more than half of the protocol's 14 transitions contain issues that prevent progress in an execution. The Listings 4.8 and 4.9 give an example of two transitions fraught with four bugs, two more transitions are discussed that contain major changes.

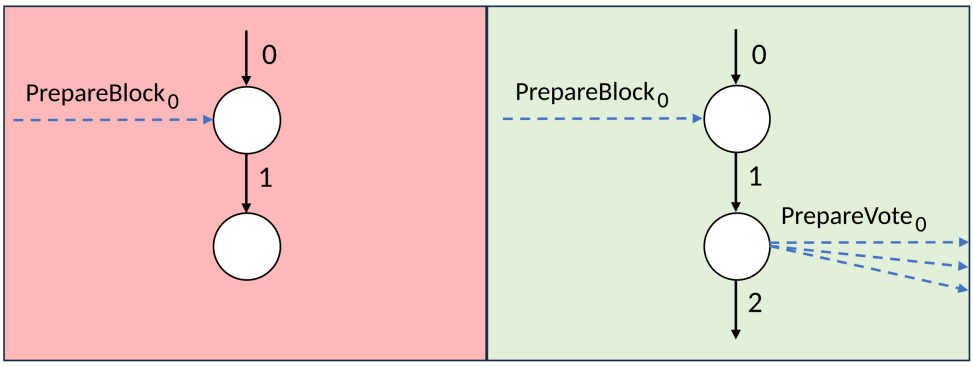

Figure 4.5: State transitions after receiving a PrepareBlock message of the first block 0. Left: the version of *prepare_block_vote* containing bugs, where a node makes no progress after receiving the PrepareBlock message of the first block. Right: after fixing the bugs in *prepare_block_vote*, can nodes successfully cast votes for blocks where the parent block already reached local prepare stage. In this special case, the Genesis Block.

The *PrepareBlock_vote* transition contains two issues. The first is that in addition to the passed in PrepareBlock message, is a vote quorum message

needed of the parent block, for the *pending_PrepareVote* function. Instead, the model code uses the PrepareBlock message of the just received block, and no additional quorum message, which leads to not even the first block being voted for. So no votes are being cast at all in the network, which means consensus can never be reached, which is illustrated in Figure 4.5.

The solution is to find the quorum message of the parent block. This needs two special cases where adhoc PrepareQC messages need to be created. One special case for the first block, as the parent block of the first block is the genesis block. As well as a special case for when the node has not received a PrepareQC message for the parent yet, but the PrepareBlock message provides the parent block as its piggyback block, which can be the case after view changes.

The second bug is that the received PrepareBlock message needs to be processed, e.g. be situated in the counting message buffer, for the *pending_PrepareVote* function. It needs the message in the counting buffer, as it checks the parent relation of the parent block with the just received PrepareBlock message's block.

```
Definition process_PrepareBlock_vote
    (s : NState) (msg : message) (s' : NState)
    (lm : list message) : Prop :=
  let quorum_msg :=
     (get_quorum_msg_for_block state (parent_of (get_block msg))
  in
  s' = (* Record outgoing PrepareVote messages *)
       record_plural
         (process s msg)
           (pending_PrepareVote s msg) /\
           (pending_PrepareVote (process s msg) quorum_msg) /\
  lm = (pending_PrepareVote s msg) /\
  lm = (pending_PrepareVote (process s msg) quorum_msg) /\
  received s msg /\
  honest_node (node_id s) /\
  get_message_type msg = PrepareBlock /\
  view_valid s msg /\
  (* PrepareBlocks cannot be processed during timeout *)
  timeout s = false /\
```

```
20  prepare_stage s (parent_of (get_block msg)).
```

Listing 4.8: Bugs in Coq model code: transition *process_PrepareBlock_vote* containing two different bugs highlighted in red, with the fix in green(local.v)[10]

The *process_PrepareQC_last_block_new_proposer* transition also contains two issues. The transition happens after a PrepareQC message has been received, three blocks reached a local quorum, and the node is the proposer for the next view, proposing the next three blocks. The main issue here is that the received PrepareQC message does not need to be the last block in this view, according to the English specification: "it is not the case that nodes necessarily vote for blocks in consecutive order" [9, Section 5.1]. Not fixing the bug, leads in this edge case to no progress being made, displayed in Figure 4.6. So the

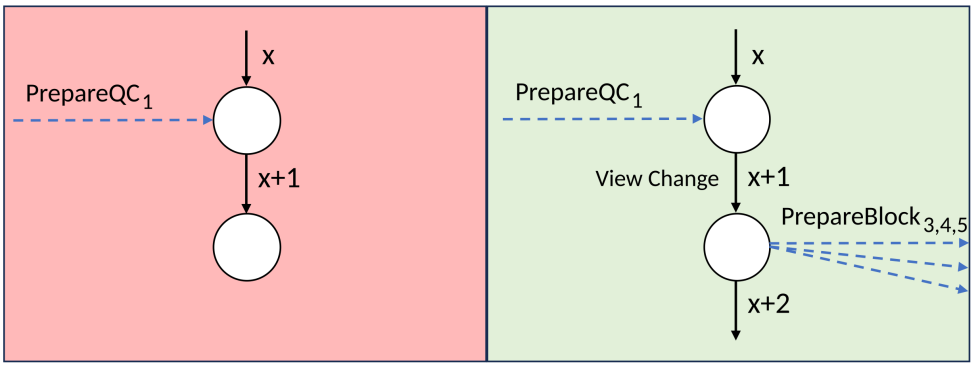

Figure 4.6: State transitions, of the next proposer, after receiving the last PrepareQC message of the current view, while the contained block, here the second block, is not the last proposed block of the view, which would be block 2. Left: the version of *process_PrepareQC_last_block_new_proposer* containing bugs, where a node makes no progress after receiving the PrepareQC message of a non-last block. Right: after removing the requirement of *last_block* and fixing the other bugs in *process_PrepareQC_last_block_new_proposer*, can the next proposer successfully change the view and propose the correct next blocks

quorum message of the last block in the view must be found, which is part of both bugs of this transition. The function *make_PrepareBlocks* needs the last block of this view, as the piggyback block for the PrepareBlock messages for the next view, as well as for the parent block of the first new block. Removing only the requirement *last_block*, while not getting the quorum message of the actual last block for the proposal of the next blocks, can, in the illustrated edge

case, cause the actual last block to get re-proposed here, resulting in an order of proposed blocks of: 0, 1, 2, 2, 3, 4.

```
Definition process_PrepareQC_last_block_new_proposer
   (s : NState) (msg : message) (s' : NState)
   (lm : list message) : Prop :=
  (* Increment the view; propose next block *)
  s' = record_plural
       (* New state *)
       (increment_view (process s msg))
         (make_PrepareBlocks (increment_view (process s msg))
         msg) ∧
         make_PrepareBlocks (increment_view (process s msg))
         quorum_msg ∧
  lm = (* Send all block proposals for the new view *)
   make_PrepareBlocks (increment_view (process s msg))
   msg ∧
   make_PrepareBlocks (increment_view (process s msg))
   quorum_msg ∧
  received s msg ∧
  honest_node (node_id s) ∧
  get_message_type msg = PrepareQC ∧
  view_valid s msg ∧
   last_block (get_block msg) ∧
  is_block_proposer (node_id s) (S (node_view s)).
```

Listing 4.9: Bugs in Coq model code: transition *process_PrepareQC_last_block_new_proposer* containing two different bugs highlighted in red, with the fix in green. (local.v)[10]

The transition *process_ViewChangeQC_single* has one condition to many. This transition models the behavior of a node receiving a ViewChangeQC message, which marks a successfull view change, as the receiving node transitions to the next view. The transition checks if a PrepareQC message has previously been received for the block contained in the ViewChangeQC message. Has the node not received it, would the transition be invalid. This is too restrictive, as through network errors could the PrepareQC message be received out of order, after the ViewChangeQC. To circumvent this, is this check removed in the protocol implementation. A more complicated protocol step could be designed, which buffers the ViewChangeQC message, until the PrepareQC has been received, or the viewchange process timeouts. This is of

no great importance though, as processing the ViewChangeQC does only lead to incrementing the view and does not handle the contained block, so the need for a PrepareQC here is not important. The carryover block quorum message can also be retrieved from the ProposeBlock message of the first next block, if the PrepareQC message has not been received at all.

The transition *process_PrepareBlock_malicious_vote* has like its honest counterpart, *process_PrepareBlock_vote*, a *pending_PrepareVote* function, see Section 4.4.3. The function checks if there are any stored PrepareBlock messages where the parent block has now reached a quorum, so a PrepareVote can now be sent out for the block in the stored PrepareBlock message. The *process_PrepareBlock_malicious_vote* transition needs its own *pending_PrepareVote* function, as it has much looser constraints, as a malicious node can cast votes for two blocks of the same height. This was not reflected in the model code, as there the ordinary *pending_PrepareVote* function was used, which prohibits this behavior.

While testing view change behavior was another problem found, a whole transition was missing, to model the view change behavior of the protocol. Listing 4.10 shows the missing transition *process_ViewChange_quorum_not_new_proposer*. Without this transition, the global transition property tests failed, as no matching model transition could be found for a valid, recorded protocol transition. This transition is for the continuity of the formal model code and is according to the English specification. The English specification mentions that ”[c]urrent block proposers, validators and to-be block proposers perform the same behaviors during” a view change process [9, Section 9.2]. The model code handles this differently, it has one transition specific for the next proposer and non for all other nodes, for the state that a quorum was reached on ViewChange messages. Inserting the transition and the specified behavior leads to non-next-proposer nodes to process the ViewChangeQC message, and manage the view change, leading to progress in the protocol during view changes. The transition does the same as the transition of the next proposer, sending a possibly pending PrepareQC message, and more importantly, a ViewChangeQC message, but does not propose three new blocks. Having all nodes send a ViewChangeQC message, leads to more robustness in the network during abnormal view changes. As the nodes won't actually change the view without it, according to the protocol. Is only the next proposer sending it, might it be lost due to various possible network issues, and the network will be stuck in the view change process for ever.

```
1 Definition process_ViewChange_quorum_not_new_proposer
```

```
2    (s: NState) (msg : Message) (s': NState)
3    (lm: list message) : Prop :=
4  let msg_vc :=
5    (highest_ViewChange_message((process state msg))) in
6  let lm_prime := []
7    :: (if ¬ (prepare_qc_already_sent state (get_block msg_vc))
8      then (makePrepareQC state msg) else [])
9    :: (make_ViewChangeQC (process state msg) msg_vc)
10 in
11 s' = (record_plural (process s msg) lm_prime)
12 lm = lm_prime ∧
13 received state msg ∧
14 honest_node (node_id s) ∧
15 get_message_type msg = ViewChange ∧
16 view_valid s msg ∧
17 view_change_quorum_in_view (process s msg) (node_view s) ∧
18 not is_block_proposer (node_id s) (S (node_view s)).
```

Listing 4.10: Missing transition in the Coq model code: transition *process_ViewChange_quorum_not_new_proposer*, for nodes that are not the next proposer. Does the same as the transition of the next proposer, sending a ViewChangeQC message for the highest block in the quorum of ViewChange messages. Also sends a possible pending PrepareQC message for the block proposed in the ViewChangeQC message

4.4.3 Bugs in the Function *pending_PrepareVote*

One liveness bug was also found in underlying functions of the transition properties, shown in Listing 4.11. This function is used several times in the protocol, for sending pending PrepareVote messages for blocks that have now reached prepare stage. The issue is in line 6, where the view of the message, of the block that might be voted for, had to be the same as the view of the quorum message of its parent block. This prevents voting for blocks starting from the second view in the network. As the first block of the second view has a different view number than its parent block, thus prevents this line the protocol from voting for it, which prevents the first block of the second view to reach a quorum. Which leads to no more votes being cast for any other following block as well. What could be checked here instead, is the view number of the current state. As there shouldn't be votes cast for blocks of past views, because this would lead to the sent PrepareVote message to-be discarded anyways.

One more bug was found in this function, which is not essential for liveness or safety but leads to unnecessary messages being sent. For example by receiving a PrepareVote should the node ”check whether it has sent a PrepareVote(b, j, v) message. If it has not, send PrepareVote(b, j, v)” [9, Section 9.1, PrepareVote], but this was not implemented in the model code. Line 14 of Listing 4.11 adds the check if a PrepareVote has already been sent by this node, which prevents multiple votes from being sent by a node for one block.

```
1  Definition pending_PrepareVote
2     (s : NState) (quorum_msg : message) : list message :=
3   map (λ prepare_block_msg ⇒
4     make_PrepareVote s quorum_msg prepare_block_msg)
5       (filter (λ msg ⇒
6          Nat.eqb (get_view msg) (get_view quorum_msg) &&
7          Nat.eqb (get_view msg) (node_view s) &&
8          negb (∃ _same_height_blockb s
9              (get_block msg)) &&
10         parent_ofb (get_block msg)
11             (get_block quorum_msg) &&
12         message_type_eqb (get_message_type msg)
13             PrepareBlock) &&
14         not prepare_vote_already_sent s (get_block msg)
15       (counting_messages s)).
```

Listing 4.11: Bug in Coq model code: *pending_PrepareVote*. The bug is highlighted in red, with the fix in green. Highlighted in grey is one nonvital fix, for not re-sending votes (local.v)[10]

Chapter 5

Results and Analysis

5.1 Test Results and Protocol Behavior

This chapter demonstrates successful protocol behavior, in the form of tables listing blocks that were proposed during executions of the instrumented integration tests. Successful, meaning the network reaches consensus on blocks in a way according to the model code of Giskard, tolerant to certain Byzantine failures. The test results show, that after the found issues were fixed, serves Giskard as a viable, Byzantine failure-tolerant consensus protocol for blockchain networks, albeit more testing is needed to prove its resistance against more complex Byzantine behavior. For a description of the protocol, see Section 2.2.4, or the specification paper [9] for a more in-depth look. The test results are accompanied by quantitative measures, that validate that every relevant transition has been compared with idealized model transitions. The irrelevant transitions, noted in the results tables, are transitions in between the model transitions, which are triggered by the nodes receiving a new message, illustrated in Figure 5.1. They are filtered out during the model transition tests, to not cause the transition tests to fail, as no model transition is defined to model these transitions' behaviors, as they are trivial and not important for the validation of Giskard. The quantitative measures also include a column stating if the safety property tests were successful or not, which fail only in the case of having a majority of malicious nodes successfully double voting.

5.1.1 Normal View Change Behavior

The most basic protocol behavior was tested first, normal view change behavior, without any Byzantine failures, no intentional timeouts, or double

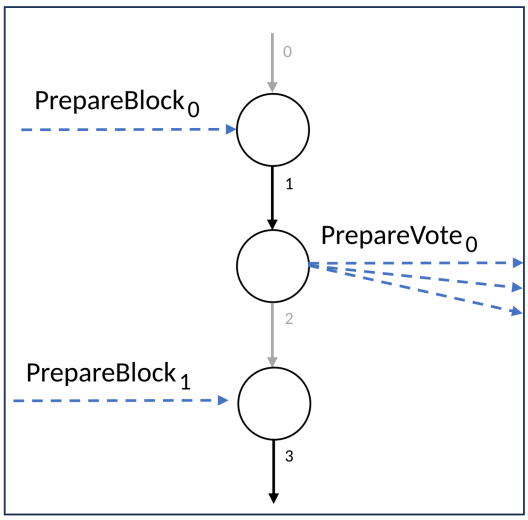

Figure 5.1: Irrelevant transitions in Giskard, which are not compared to model transitions, marked in grey. No model transitions exist for them, as the transitions are just the nodes receiving the next message, including the initial transition.

voting. Results can be seen in Table 5.1, which is also illustrated in Figure 4.3 a). Nine blocks were proposed, across three views, each block received a vote by all four participating nodes, and thus, all blocks reached a quorum. Each participating node had a local quorum of every block, which can be seen in the column *Prepare-QC*, as four PrepareQC messages were sent out for every block. Three blocks were proposed per view, for three views with consecutive view numbers, which indicates that no abnormal view change through a timeout happened.

5.1.2 Abnormal View Change Behavior

Crash tolerance is achieved with abnormal view change behavior in the protocol. The most severe crashes for reaching consensus is when the proposer crashes before it could propose all blocks of a view. In the implementation, if a non-proposing node is crashing, can other nodes detect this through a lost tcp connection, which will only lead to fewer nodes participating in the current and future views, so fewer votes are necessary to reach a quorum. It is not fully implemented yet, nor tested, as it was not part of the model code, and may be

Test Results for normal View Change: no timeouts								
Block Height	View	Block Index	Block id	Proposer	Votes	Prepare QC	Honest	Carry-over
0	0	1	609df0	021df4	4	4	yes	no
1	0	2	5cfc24	021df4	4	4	yes	no
2	0	3	14caa4	021df4	4	4	yes	no
3	1	1	9bfcf8	02acb3	4	4	yes	no
4	1	2	37abcb	02acb3	4	4	yes	no
5	1	3	c24e6f	02acb3	4	4	yes	no
6	2	1	4c71f6	02b216	4	4	yes	no
7	2	2	6fc81b	02b216	4	4	yes	no
8	2	3	29f261	02b216	4	4	yes	no

Quantitative measures				
Safety Tests Passed	Total Transitions	Irrelevant Transitions	Tested Transitions	Untested Transitions
yes	304	152	152	0

Table 5.1: Test results of the normal view change test scenario. After three blocks have reached local prepare stage in a node, does it automatically perform a normal view change, by incrementing its view. Is it the proposer for the next view, does it wait until it has received three new blocks from the network and proposes them.

interesting for future work. The more disrupting cases where a proposer is crashing during block proposal are explored in the next three tests.

Failure detection of timeouts is handled rudimentary in this implementation and only relies on the local clock of the machine that all nodes are running on. No extra protocol, like a heartbeat protocol, see Section 2.2.1.2, has been written, to adjust node clock times between each other. This is as the tests are only run on a single machine. In addition to that, are all timeout tests implemented to happen controlled and only once. A timeout is triggered when the time for a view ran out. The timeout flag in the local state is set to true, and the nodes exchange ViewChange messages with their local highest block in prepare stage, e.g. their highest block that reached a vote quorum. As soon as there is a quorum with the amount of ViewChange messages, the nodes select the highest block of those messages, and send a PrepareQC, as well as a ViewChangeQC message for this block, which triggers the ViewChange. The carryover blocks are marked in the last column with a *c*.

Table 5.2, shows behavior similar to Figure 4.4, where the proposer of the second view proposes no blocks. The last block of the first view reached prepare stage in all nodes and got correctly selected as the carryover block. As the view with view number one timed out, and no block was proposed, continue blocks 3 to 5 with view number two and not one.

Test Results for abnormal View Change: no block proposed								
Block Height	View	Block Index	Block id	Proposer	Votes	Prepare QC	Honest	Carry-over
0	0	1	dff5a0	020fbc	4	4	yes	no
1	0	2	10bcf1	020fbc	4	4	yes	no
2	0	3	f745b7	020fbc	4	5	yes	yes
3	2	1	783cc0	0367b9	4	4	yes	no
4	2	2	b352e7	0367b9	4	4	yes	no
5	2	3	c5d749	0367b9	4	4	yes	no

Quantitative measures				
Safety Tests Passed	Total Transitions	Irrelevant Transitions	Tested Transitions	Untested Transitions
yes	343	173	170	0

Table 5.2: Test results of the abnormal view change test scenario, where no block has been proposed by the next proposer. Timeout of the proposer before it could propose a single block in view 1. Exchange of ViewChange messages with the highest block in prepare stage being block 2. The proposer for view 2 took over and proposed the next three blocks.

The next test case shows the behavior of the protocol where the first block of a view was the highest block in local prepare stage, see Table 5.3. The timeout was triggered by the test right after the first block reaches a local quorum, which also sent a PrepareVote for the next block. This is not received by any node, as the timeout is already triggered, so the second block still did not reach a quorum. Timeouts let nodes discard messages apart from ViewChange, ViewChangeQC, and PrepareQC messages, so no further block could reach a quorum after a node has already sent its ViewChange message with its highest block. This can be seen by block 4 and 5 in view 1, as they haven't received any PrepareQC messages. The first block of view 1 got successfully selected as the carryover block, the view changes and the blocks 4 and 5 of view 1 get re-proposed by the next proposer in view 2.

Table 5.4 shows the behavior displayed in Figure 4.3 b). This is a special

Test Results for abnormal View Change: first block global prepare stage								
Block Height	View	Block Index	Block id	Proposer	Votes	Prepare QC	Honest	Carry-over
0	0	1	4adf74	028bcb	4	4	yes	no
1	0	2	e44c1b	028bcb	4	4	yes	no
2	0	3	e80a39	028bcb	4	4	yes	no
3	1	1	c65b5a	02ad17	4	4	yes	yes
4	1	2	f44503	02ad17	4	0	yes	no
5	1	3	161931	02ad17	0	0	yes	no
4	2	1	f44503	038781	4	4	yes	no
5	2	2	161931	038781	4	4	yes	no
6	2	3	64becf	038781	4	4	yes	no

Quantitative measures				
Safety Tests Passed	Total Transitions	Irrelevant Transitions	Tested Transitions	Untested Transitions
yes	408	206	202	0

Table 5.3: Test results of the abnormal view change test scenario, where the first block reached global prepare stage. Block 3, the first block of view 1, reached global prepare stage before a timeout occurred. The votes for block 4 were sent but ignored, due to the timeout happening right before, so block 4 did not reach local prepare stage in any block.

case, where two different same-height blocks reach local prepare stage. This does not violate the safety properties, as both blocks reach prepare stage in different views and not globally, which means that the network still has a consistent blockchain, and does not contain two same height blocks. This test is specifically made for this scenario to happen, all nodes but one are instructed to trigger their timeout when the first block of view 1 reached a quorum. The other node receives all votes for the second block of view 1, which leads to the second block reaching a local quorum in it. This can be confirmed in the table, as a PrepareQC message was sent for block 4 and for block 5 a PrepareVote, as its child block reached local prepare stage. After that, does it also trigger a timeout, and does not send a ViewChange message, so the second block does not get selected during the view change process. This can be seen in the table as well, as the first block of view 1 gets selected as the carryover block and not the second.

Test Results for abnormal View Change: second block local prepare stage								
Block Height	View	Block Index	Block id	Proposer	Votes	Prepare QC	Honest	Carry-over
0	0	1	fd3b3a	022950	4	4	yes	no
1	0	2	5d569c	022950	4	4	yes	no
2	0	3	379d8c	022950	4	4	yes	no
3	1	1	bc2e59	030fbc	4	4	yes	yes
4	1	2	c27c40	030fbc	4	1	yes	no
5	1	3	029a66	030fbc	1	0	yes	no
4	2	1	c27c40	0341fa	4	4	yes	no
5	2	2	029a66	0341fa	4	4	yes	no
6	2	3	602cd7	0341fa	4	4	yes	no

Quantitative measures				
Safety Tests Passed	Total Transitions	Irrelevant Transitions	Tested Transitions	Untested Transitions
yes	410	207	203	0

Table 5.4: Test results of the abnormal view change test scenario, where the second block has reached local prepare stage in one crashing node. Block 4, the second block of view 1, reached local prepare stage in one node, which it couldn't announce to the others. This results in two blocks of same height reaching local prepare stage, as this is in different views, is the first safety property not violated and the safety tests still succeed.

5.1.3 Malicious Behavior

Byzantine failures not only include timeouts or crashes but also malicious behavior, be it deliberate or not. The primary form of Byzantine behavior Giskard considers is double voting, sending two PrepareVote messages for two different blocks of the same height within the same view [10, local.v]. In the following, are two test scenarios shown in Tables 5.5, and 5.6, to show failing the safety property tests, while showing the resilience of Giskard against most basic malicious Byzantine behavior. The theoretical upper boundary for Byzantine failure tolerance is the aforementioned third of participating nodes in a vote [40]. For this reason, are four nodes used in both tests, so there can be a test case with one malicious node and three honest nodes, being less than a third of the participating nodes, thus being not successful in overtaking the network with its proposed extra block. This can be seen in the table, as there is only one vote cast for the malicious block, which is marked with an x in

the *honest* column. In Table 5.6 on the other hand, cast the three dishonest nodes 3 votes for the malicious block, reaching a global quorum on it, at the same time as the honest block of the same height. This leads to the safety property tests failing, which is shown in the column *Safety Tests Passed*. A test showing the protocol fail at the one-third boundary was not implemented, as this involves more complex behavior, tricking honest nodes into voting for the malicious block, but could be done in future work. The paper with the specification of Giskard, proposes aggregated signatures to be used to prevent spoofing [9], similar to Lamport's authenticators, see Section 2.2.1.2. Implementing this, and the malicious node behavior needed to spoof those signatures is needed, to show the protocol's Byzantine failure boundary of a third. In a blockchain network would then one third of malicious nodes be needed, being able to spoof aggregated signatures before the proposer could send out its PrepareBlock message, to convince honest nodes that the real proposer proposed the malicious block, and trick the nodes into voting for the malicious block instead.

In the basic malicious behavior tests, is a malicious block proposed, with the proposer named *"NotTrustworthy"*, and the payload being *"Beware, I am a malicious block"*, this lets the malicious nodes identify this block as the malicious block to double vote for. Honest nodes discard all messages containing this block, as the proposer is not the proposer of the current view, which is the reason why in the instrumented tests, a two third majority of malicious nodes is needed, for the block to reach global prepare stage.

5.2 Discussion of the Results

The test results show that an executable implementation of Giskard works under normal network conditions, as well as under crash and Byzantine failures, during the timeout and malicious node behavior test scenarios. The implementation uncovered liveness bugs in the formal model code [10], which needed to be fixed for the protocol to work and reach consensus. As of the writing of this thesis, could the formally verified code not be adapted to these updates, so the updated protocol works in practice, but is not yet formally verified. The proposed updates are though according to the English specification [9]. Thus, the test results suggest that Giskard serves as a viable consensus protocol for blockchain networks. This matters, as viable alternatives to predominant consensus protocols, e.g. Bitcoin's PoW, Ethereum's, or Algorand's PoS protocols, are continuously explored in the scientific community, and in companies. As many issues still exist in

Test Results for one dishonest node: double voting a malicious block								
Block Height	View	Block Index	Block id	Proposer	Votes	Prepare QC	Honest	Carry-over
0	0	1	c43b6a	02232f	4	4	yes	no
1	0	2	899720	02232f	4	4	yes	no
2	0	3	fd0b28	02232f	4	4	yes	no
3	1	1	bf1f63	026a06	4	4	yes	no
4	1	2	7432ea	026a06	4	4	yes	no
5	1	3	19bd95	026a06	4	4	yes	no
3	1	1	bf1f63	NotTru	1	0	no	no
6	2	1	16612f	027b87	4	4	yes	no
7	2	2	e2c1a1	027b87	4	4	yes	no
8	2	3	9f888e	027b87	4	4	yes	no

Quantitative measures				
Safety Tests Passed	Total Transitions	Irrelevant Transitions	Tested Transitions	Untested Transitions
yes	306	153	153	0

Table 5.5: Test results of the malicious behavior scenario, with one malicious node. One dishonest node double-voted for the blocks of height 3. The malicious block did not reach a quorum, so its transactions are not included in the global chain of blocks, not breaking its integrity, and the safety tests still succeed.

blockchain networks, like the correctness of used code, energy consumption, the throughput of the number of transactions per minute, security against malicious actors, or safety against scams. The test results don't show however, if Giskard is in any regard better or worse than any other protocol, which should be investigated in future work. They only show that the protocol can work in a simulated blockchain network, to reach consensus on blocks in a blockchain network, under some specific crash or Byzantine failures. Furthermore, the ultimate test to show that Giskard is a viable consensus protocol can only be shown when it is used in an actual live network. A network with thousands of users, over a span of several years, where more bugs might be found. Especially, when it is tried to actively disrupt the network, like with any other blockchain network that proved its resilience over the years. A more detailed approach to possible future work is given in Section 6.2.

Test Results for three dishonest nodes: double voting a malicious block								
Block Height	View	Block Index	Block id	Proposer	Votes	Prepare QC	Honest	Carry-over
0	0	1	5932a8	02bac6	4	4	yes	no
1	0	2	8c7779	02bac6	4	4	yes	no
2	0	3	f42f77	02bac6	4	4	yes	no
3	1	1	9edbf4	02d3e4	4	4	yes	no
4	1	2	23f640	02d3e4	4	4	yes	no
5	1	3	648f81	02d3e4	4	4	yes	no
3	1	1	9edbf4	NotTru	3	3	no	no
6	2	1	5fcd91	02ec64	4	4	yes	no
7	2	2	18cbd1	02ec64	4	4	yes	no
8	2	3	d621cb	02ec64	4	4	yes	no

Quantitative measures				
Safety Tests Passed	Total Transitions	Irrelevant Transitions	Tested Transitions	Untested Transitions
no	340	170	170	0

Table 5.6: Test results of the malicious behavior scenario, with three malicious nodes. Three dishonest nodes double-voted for the blocks of height 3, which both reached global prepare stage. Thus, its transactions are included in the global chain of blocks, breaking the integrity of the blockchain, and the safety tests fail.

Chapter 6

Conclusions and Future Work

Based on the evidence of the test results of this thesis, which can be accessed on Zenodo [71], serves Giskard as a viable consensus protocol for blockchain networks. It can reach consensus on a chain of blocks successfully, see Chapter 5. Even with the occurrence of simple crash and other Byzantine failures, for which a two-third majority of honest nodes is needed.

However, the protocol cannot be exactly implemented as the formal model code [10]. Without the suggested updates from Section 4.4 can no substantial progress be made, and no consensus can be reached on a chain of blocks. Until all formal model code, including proof scripts, are fully updated, the new implementation proposed in this thesis cannot be considered formally verified, and this is left as future work. The updated implementation, on the other hand, is according to the English specification [9], which also serves as a valid specification for Giskard, yet with more room for interpretation than the model code.

To come to this conclusion, was prior model code of Giskard translated to an executable language, Python, and integrated into a blockchain network framework, called Hyperledger Sawtooth. An integration test is run, and all local states of participating nodes in the network are recorded for later comparison with expected model behavior, which is described in Chapter 4. During the integration tests receive the nodes new blocks from the network, with the help of Giskard try they to reach consensus on whether they should include a block in the blockchain or not.

The results from the orchestrated tests show, that under normal network behavior, the protocol executions are according to the updated specification and the network reaches consensus, see Chapter 5. Specifically designed tests were designed to show the protocol's behavior under certain abnormal

Byzantine behavior, where the network using Giskard also proves to behave exactly like the updated model code.

6.1 Answering the Research Questions

No, the protocol cannot be implemented exactly as the given Coq model code [10] without some updates, which are in accordance to the mathematical English specification [9], to run in a simulated distributed ledger.

Yes, an implementation of Giskard in accordance to an updated version of the model code, can achieve consensus in the presence of Byzantine faults, while fulfilling the specified, abstract, safety properties.

6.2 Future Work

Further work is needed, to prove the upper bounds of Giskard's Byzantine failure tolerance. As well as bringing the implementation to a level, so that it can be used in a live test network, where more deliberate acts of disrupting the network, might uncover more bugs in the protocol, or even a lower failure tolerance than expected.

But first, should the Coq model code be updated with the proposed fixes, and the proofs re-run, so the model still verifies the protocol. This should be done, as others might use the Coq code as a basis for their own implementations of Giskard, so it should be bug-free as much as possible.

Bringing the implementation to a level where it can be used in a live test network, includes several things to be implemented in the future. At first, should the network send several blocks to the consensus engine, instead of only one by one, which would also eliminate the need for the PoET consensus protocol being used in the background for satisfying the Sawtooth network, also explained in Section 4.2.

Second, could an epoch change be implemented, so the network can run more nodes than necessary for a view, and more protocol behavior can be tested. For this, the model code needs to be updated, as it does not include epoch change behavior as of the time of this writing. With the epoch change should also a proper timeout protocol be implemented for triggering abnormal view changes, as well as a proof of stake-based selection protocol for selecting nodes for an epoch.

Third, for testing a network with more than four nodes, needs the GiskardTester to be updated. As shown in Section 4.3, does its process get killed by the operating system for running out of memory. Tests that record less data could make global transition tests possible, running thousands of nodes, which is the ultimate test showing that Giskard is a viable protocol for running in a realistic blockchain network.

Also interesting would be a comparison between the two Sawtooth consensus algorithms and Giskard, in time to reach consensus, as well as amount of data/msgs sent, or energy consumption.

Proving the upper bounds of Byzantine failure tolerance further is explained roughly in Section 5.1.3.

References

[1] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *J. ACM*, vol. 32, no. 2, pp. 374–382, 1985. doi: https://doi.org/10.1145/3149.214121 [Pages 1, 13, and 14.]

[2] E. Shi, *Foundations of Distributed Consensus and Blockchains*. Preliminary Draft, 2020, http://elaineshi.com/docs/blockchain-book.pdf. [Page 1.]

[3] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008. [Pages 1, 6, 8, 9, 10, and 26.]

[4] M. A. Alturki, J. Chen, V. Luchangco, B. Moore, K. Palmskog, L. Peña, and G. Roşu, "Towards a verified model of the Algorand consensus protocol in Coq," in *Formal Methods 2019 International Workshops*, 2020. doi: https://doi.org/10.1007/978-3-030-54994-7_27 pp. 362–367. [Page 1.]

[5] G. Pîrlea and I. Sergey, "Mechanising blockchain consensus," in *Certified Programs and Proofs*, 2018. doi: https://doi.org/10.1145/3167086 pp. 78–90. [Page 1.]

[6] S. E. Thomsen and B. Spitters, "Formalizing Nakamoto-style proof of stake," in *Computer Security Foundations Symposium*, 2021. doi: https://doi.org/10.1109/CSF51468.2021.00042 pp. 1–15. [Page 1.]

[7] P. Fonseca, K. Zhang, X. Wang, and A. Krishnamurthy, "An empirical study on the correctness of formally verified distributed systems," in *European Conference on Computer Systems*, 2017. doi: https://doi.org/10.1145/3064176.3064183 pp. 328–343. [Page 1.]

[8] T. D. Chandra, R. Griesemer, and J. Redstone, "Paxos made live - an engineering perspective," in *Symposium on Principles of Distributed*

Computing, 2007. doi: https://doi.org/10.1145/1281100.1281103 [Page 1.]

[9] E. Li, K. Palmskog, M. Sebe, and G. Roşu, "Specification of the Giskard consensus protocol," 2020. [Pages 2, 3, 4, 17, 19, 23, 28, 36, 41, 44, 47, 49, 51, 52, 58, 61, and 62.]

[10] E. Li, K. Palmskog, and M. Sebe, "Giskard consensus protocol Coq code version 1.1." [Online]. Available: https://github.com/runtimeverification/giskard-verification/releases/tag/v1.1 [Pages xi, xii, 2, 19, 20, 22, 24, 32, 33, 34, 43, 44, 45, 47, 48, 51, 57, 58, 61, and 62.]

[11] E. Li, K. Palmskog, M. Sebe, and G. Roşu, "Verifying safety of the Giskard consensus protocol in Coq," 2020, unpublished. [Online]. Available: https://github.com/runtimeverification/giskard-verification/releases/download/v1.0/report.pdf [Pages 2, 3, 4, 19, and 23.]

[12] PlatON Network, "PlatON website," 2020, https://platon.network/en. [Page 3.]

[13] D. L. Stearns, *Electronic value exchange: Origins of the VISA electronic payment system*. Springer, 2011. [Page 5.]

[14] Z. B. Omariba, N. B. Masese, and G. Wanyembi, "Security and privacy of electronic banking," *International Journal of Computer Science Issues (IJCSI)*, vol. 9, no. 4, p. 432, 2012. [Page 5.]

[15] A. Murphy, "An analysis of the financial crisis of 2008: Causes and solutions," *SSRN Electronic Journal*, 2008. doi: https://doi.org/10.2139/ssrn.1295344 [Page 5.]

[16] W. Li, C. Feng, L. Zhang, H. Xu, B. Cao, and M. A. Imran, "A scalable multi-layer pbft consensus for blockchain," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 5, pp. 1146–1160, 2021. doi: https://doi.org/10.1109/TPDS.2020.3042392 [Page 6.]

[17] H. A. Cryptography and Society. Robustness and efficiency of consensus protocols. [Online]. Available: https://projects.iq.harvard.edu/applied-cryptography-society/program-robustness_and_efficiency_consensus_protocols [Page 6.]

[18] B. Vitalik. Ethereum whitepaper | ethereum.org. [Online]. Available: https://ethereum.org/en/whitepaper/ [Pages 6 and 26.]

[19] S. Micali, "ALGORAND: the efficient and democratic ledger," *CoRR*, vol. abs/1607.01341, 2016. [Online]. Available: http://arxiv.org/abs/1607.01341 [Page 6.]

[20] A. Brock, D. Atkinson, E. Friedman, E. Harris-Braun, E. Mcguire, J. M. Russell, N. Perrin, N. Luck, and W. Harris-Braun, "Holo green paper," *Green Paper*, 2018. [Pages 6 and 7.]

[21] S. Popov and Q. Lu, "Iota: feeless and free," *IEEE Blockchain Technical Briefs*, 2019. [Pages 6 and 7.]

[22] K. Crary, "Verifying the hashgraph consensus algorithm," 2021, unpublished. [Online]. Available: https://www.cs.cmu.edu/~crary/papers/2021/hashgraph.pdf [Page 6.]

[23] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. D. Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, S. Muralidharan, C. Murthy, B. Nguyen, M. Sethi, G. Singh, K. Smith, A. Sorniotti, C. Stathakopoulou, M. Vukolić, S. W. Cocco, and J. Yellick, "Hyperledger fabric," in *Proceedings of the Thirteenth EuroSys Conference*. ACM, apr 2018. doi: https://doi.org/10.1145/3190508.3190538 [Pages 6 and 26.]

[24] A deep dive into blockchain scalability. [Online]. Available: https://crypto.com/university/blockchain-scalability [Page 6.]

[25] Algorand boosts performance 5x in latest upgrade - algorand developer portal. [Online]. Available: https://developer.algorand.org/articles/algorand-boosts-performance-5x-in-latest-upgrade/ [Page 6.]

[26] Benchmarking cassandra scalability on AWS — over a million writes per second | by netflix technology blog | netflix TechBlog. [Online]. Available: https://netflixtechblog.com/benchmarking-cassandra-scalability-on-aws-over-a-million-writes-per-second-39f45f066c9e [Page 6.]

[27] Visa. Small business retail | visa. [Online]. Available: https://usa.visa.com/run-your-business/small-business-tools/retail.html [Page 6.]

[28] K. Birman, "The promise, and limitations, of gossip protocols," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 5, p. 8–13, oct 2007. doi: https://doi.org/10.1145/1317379.1317382 [Page 7.]

[29] L. Lamport, “Password authentication with insecure communication,” *Commun. ACM*, vol. 24, no. 11, p. 770–772, nov 1981. doi: https://doi.org/10.1145/358790.358797 [Page 8.]

[30] A. Kirti and V. H. K., “Hash_rc6 - variable length hash algorithm using rc6,” in *2015 International Conference on Advances in Computer Engineering and Applications*, 2015. doi: https://doi.org/10.1109/ICACEA.2015.7164747 pp. 450–456. [Pages 8 and 9.]

[31] A. Brownworth, “Blockchain demo,” original-date: 2016-11-04. [Online]. Available: https://github.com/anders94/blockchain-demo [Page 9.]

[32] R. C. Merkle, “Method of providing digital signatures,” USA patentus 4 309 569A, 1982. [Online]. Available: https://patents.google.com/patent/US4309569/en [Page 9.]

[33] L. Lamport, R. Shostak, and M. Pease, “The byzantine generals problem,” *ACM Trans. Program. Lang. Syst.*, vol. 4, no. 3, p. 382–401, jul 1982. doi: https://doi.org/10.1145/357172.357176 [Page 9.]

[34] C. E. TREVATHAN, T. D. TAYLOR, R. G. HARTENSTEIN, A. C. MERWARTH, and W. N. STEWART, “Development and application of nasa’s first standard spacecraft computer,” *Communications of the ACM*, vol. 27, 1984. [Page 10.]

[35] Y. Yeh, “Triple-triple redundant 777 primary flight computer,” in *1996 IEEE Aerospace Applications Conference. Proceedings*, vol. 1, 1996. doi: https://doi.org/10.1109/AERO.1996.495891 pp. 293–307 vol.1. [Page 10.]

[36] T. Ishigooka, S. Honda, and H. Takada, “Cost-effective redundancy approach for fail-operational autonomous driving system,” in *2018 IEEE 21st International Symposium on Real-Time Distributed Computing (ISORC)*, 2018. doi: https://doi.org/10.1109/ISORC.2018.00023 pp. 107–115. [Page 10.]

[37] T. Haerder and A. Reuter, “Principles of transaction-oriented database recovery,” *ACM Comput. Surv.*, vol. 15, no. 4, p. 287–317, dec 1983. doi: https://doi.org/10.1145/289.291 [Pages 10 and 11.]

[38] M. Herlihy, N. Shavit, V. Luchangco, and M. Spear, *The Art of Multiprocessor Programming*. Newnes, 2020. [Page 11.]

[39] G. Coulouris, J. Dollimore, T. Kindberg, and G. Blair, "Distributed systems concepts and design 5th ed," 2012. [Pages 11, 12, and 14.]

[40] M. J. Fischer, "The consensus problem in unreliable distributed systems (a brief survey)," in *Foundations of Computation Theory*, M. Karpinski, Ed. Springer Berlin Heidelberg, 1983. ISBN 978-3-540-38682-7 pp. 127–140. [Pages 12, 13, 14, and 57.]

[41] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm (extended version)," in *Proceeding of USENIX annual technical conference, USENIX ATC*, 2014, pp. 19–20. [Pages 13 and 15.]

[42] L. Lamport, *Time, Clocks, and the Ordering of Events in a Distributed System*. Communications of the ACM, July 1978, vol. 21, no. 7. ISBN 9781450372701 [Pages 13 and 14.]

[43] M. Pease, R. Shostak, and L. Lamport, "Reaching agreement in the presence of faults," *J. ACM*, vol. 27, no. 2, p. 228–234, apr 1980. doi: https://doi.org/10.1145/322186.322188 [Page 13.]

[44] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, vol. 21, no. 2, pp. 120–126, 1978. [Page 13.]

[45] ——, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, vol. 21, no. 2, pp. 120–126, 1978. [Page 13.]

[46] L. Lamport, "The part-time parliament," *ACM Trans. Comput. Syst.*, vol. 16, no. 2, pp. 133–169, 1998. doi: https://doi.org/10.1145/279227.279229 [Page 14.]

[47] ——, "Paxos made simple," *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pp. 51–58, December 2001. [Online]. Available: https://www.microsoft.com/en-us/research/publication/paxos-made-simple/ [Page 15.]

[48] M. Castro, B. Liskov *et al.*, "Practical byzantine fault tolerance," in *OsDI*, vol. 99, no. 1999, 1999, pp. 173–186. [Pages 15 and 16.]

[49] E. Deirmentzoglou, G. Papakyriakopoulos, and C. Patsakis, "A survey on long-range attacks for proof of stake protocols," *IEEE Access*, vol. 7,

pp. 28 712–28 725, 2019. doi: https://doi.org/10.1109/ACCESS.2019.2901858 [Pages 16 and 17.]

[50] K. Olson, M. Bowman, J. Mitchell, S. Amundson, D. Middleton, and C. Montgomery, "Sawtooth: An introduction," 2018. [Pages 16 and 26.]

[51] Proof of work vs proof of stake: Which is better? (PoS vs PoW). [Online]. Available: https://coindcx.com/blog/crypto-basics/proof-of-work-vs-proof-of-stake/ [Page 16.]

[52] A. de Vries, "Bitcoin's energy consumption is underestimated: A market dynamics approach," *Energy Research & Social Science*, vol. 70, p. 101721, 2020. doi: https://doi.org/10.1016/j.erss.2020.101721 [Pages 16 and 17.]

[53] B. Alpern and F. B. Schneider, "Defining liveness," *Information Processing Letters*, vol. 21, no. 4, pp. 181–185, 1985. doi: https://doi.org/10.1016/0020-0190(85)90056-0. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0020019085900560 [Page 17.]

[54] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham, "Hotstuff: Bft consensus in the lens of blockchain," *arXiv preprint arXiv:1803.05069*, 2018. [Page 18.]

[55] I. Sergey, J. R. Wilcox, and Z. Tatlock, "Programming and proving with distributed protocols," *PACMPL*, vol. 2, no. POPL, pp. 28:1–28:30, 2018. doi: https://doi.org/10.1145/3158116 [Page 22.]

[56] hyajam, "Just another blockchain simulator," original-date: 2021-04-23T23:33:45Z. [Online]. Available: https://github.com/hyajam/jabs [Pages 23 and 26.]

[57] "Scorex 2 - the modular blockchain framework," original-date: 2016-06-28T12:13:40Z. [Online]. Available: https://github.com/hyperledger-labs/Scorex [Pages 23 and 26.]

[58] P.-L. Wang, T.-W. Chao, C.-C. Wu, and H.-C. Hsiao, "Tool: An efficient and flexible simulator for byzantine fault-tolerant protocols," in *2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2021. doi: https://doi.org/10.1109/DSN53405.2022.00038. ISBN 978-1-66541-693-1 pp. 287–294. [Pages 23 and 26.]

[59] C. Faria, "BlockSim: Blockchain simulator," original-date: 2018-03-12T11:34:18Z. [Online]. Available: https://github.com/carlosfaria94/blocksim [Pages 23 and 26.]

[60] C. Berger, S. B. Toumia, and H. P. Reiser, "Simulating BFT protocol implementations at scale." [Page 23.]

[61] "VIBES: Fast blockchain simulations for large-scale peer-to-peer networks," original-date: 2018-03-15T15:00:04Z. [Online]. Available: https://github.com/i13-msrg/vibes [Pages 23 and 26.]

[62] E. Bartocci, Y. Falcone, A. Francalanza, and G. Reger, *Introduction to Runtime Verification*. Springer International Publishing, 2018, pp. 1–33. ISBN 978-3-319-75632-5 [Page 24.]

[63] M. Utting and B. Legeard, *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufman, 01 2007. ISBN 978-0-12-372501-1 [Page 24.]

[64] M. Abadi and L. Lamport, "The existence of refinement mappings," in *1988 Proceedings. Third Annual Symposium on Logic in Computer Science*, 1988. doi: https://doi.org/10.1109/LICS.1988.5115 pp. 165–175. [Page 25.]

[65] Why substrate? | substrate_ docs. [Online]. Available: https://docs.substrate.io/fundamentals/why-substrate/ [Page 26.]

[66] G. L. Fenves, "Object-oriented programming for engineering misc development," *Engineering with Computers*, vol. 6, no. 1, pp. 1–15, Dec 1990. doi: https://doi.org/10.1007/BF01200200 [Page 26.]

[67] The beacon chain | ethereum.org. [Online]. Available: https://ethereum.org/en/upgrades/beacon-chain/ [Page 27.]

[68] ZeroMQ. [Online]. Available: https://zeromq.org/ [Page 28.]

[69] Sawtooth architecture guide. [Online]. Available: https://sawtooth.hyperledger.org/docs/1.2/architecture/ [Page 28.]

[70] IOAutomata. [Online]. Available: https://www.cs.yale.edu/homes/aspnes/pinewiki/IOAutomata.html [Page 27.]

[71] L. Sandner and K. Palmskog, "sawtooth-giskard," Aug. 2023, installation instructions in the readme file. [Online]. Available: https://doi.org/10.5281/zenodo.8272100 [Pages xi, 28, 29, 30, 33, 34, 35, 36, 38, 40, 44, and 61.]

[72] Setting up a sawtooth network. [Online]. Available: https://sawtooth.hyperledger.org/docs/1.2/sysadmin_guide/setting_up_sawtooth_network.html [Page 29.]

[73] M. Giampietro and K. Mayumi, "Unraveling the complexity of the jevons paradox: The link between innovation, efficiency, and sustainability," *Frontiers in Energy Research*, vol. 6, 2018. doi: https://doi.org/10.3389/fenrg.2018.00026 [Page 30.]