UNIVERSITY OF GRONINGEN

FACULTY OF SCIENCE AND ENGINEERING

Permissioned Blockchains: A Comparative Study

A Deep Dive into Hyperledger Fabric and Hyperledger Besu

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR DEGREE OF MASTER OF SCIENCE IN COMPUTING SCIENCE

Written by:
Mark Soelman

Under the supervision of:
Arnaud VAN GELDER (IBM) **Vasilios ANDRIKOPOULOS (UG)**
Moazzem HOSSEN (IBM) **Dimka KARASTOYANOVA (UG)**

June, 2021

Acknowledgements

I would like to express my gratitude to Arnaud van Gelder, Moazzem Hossen, Vasilios Andrikopoulos, Arne Rutjes, Dimka Karastoyanova and other colleagues at the IBM CIC Groningen for their guidance, assistance, and reviews, which were valuable for the completion of this thesis. Thank you.

Abstract

Public blockchain networks such as Ethereum are typically not suitable for enterprise use cases due to varying reasons such as their low throughput, fees, inefficient consensus protocols and inability to store sensitive data confidentially. Permissioned blockchain frameworks allow enterprises to set up a private network to enable trustless collaborations securely. Hyperledger Fabric and Hyperledger Besu are two such frameworks. The aim of this thesis is to provide an in-depth comparison of both frameworks.

More specifically, this study looked at their architectures and architectural components, their transaction flows from transaction creation until commit, how both frameworks reach agreement on ledger state using consensus protocols, how each framework aims to achieve data privacy within the network, and how each framework performs in practice for various configurations. For the first four perspectives, both frameworks are first analyzed separately, after which their differences are discussed side-by-side. Our findings are both of a higher level of abstraction, as well as a more fine-grained technical level.

Some of our findings on the architecture include that Fabric uses infrastructural nodes, is highly configurable, is programming language-agnostic, and supports off-chain identity management. Besu is much simpler in most aspects including smart contract design, network composition, operations and configurability. Fabric's transaction flow is optimized for efficiency but is prone to manipulation when configured insecurely, whereas Besu's transaction flow is less efficient but leaves no room for ledger manipulation. Similarly, Besu's consensus protocols tolerate a degree of adversarial participants, whereas Fabric requires all consensus nodes to be honest. Fabric's data privacy is much more flexible than Besu's, but should be used with care. Various issues with Besu's implementation of data privacy were identified. In terms of benchmarking, both frameworks could be tweaked for a stable throughput of at least 320 transactions per second on a desktop computer. Fabric nodes are adapting to the network load, whereas Besu should be carefully tailored for an expected throughput. For both frameworks, a number of ideas for future research are provided that could potentially enhance both frameworks.

Table of contents

List of Abbreviations

ABAC Attribute-Based Access Control. 59, 113, 115

ABI Application Binary Interface. 72, 73

API Application Programming Interface. 16, 30, 31, 43, 48, 52, 55, 85, 86, 148

BFT Byzantine Fault Tolerance. 11, 90, 91, 97, 102, 104, 105, 110, 148, 149

BNF Backus-Naur Form. 164

CA Certificate Authority. 22, 24, 25, 50, 62, 164

CFT Crash Fault Tolerance. 90, 91, 97, 109, 110

CPU Central Processing Unit. 15, 92, 98, 130, 135, 144, 150, 155

DAG Directed Acyclic Graph. 6, 8, 9, 28

DLT Distributed Ledger Technology. 5, 6, 8–10, 13–15, 17, 28, 56, 61, 87, 109

DSL Domain Specific Language. 17, 42, 57, 148

ECDSA Elliptic Curve Digital Signature Algorithm. 37, 40, 50

EEA Enterprise Ethereum Alliance. 18, 45, 48, 49, 58, 59, 119

EIP Ethereum Improvement Proposal. 53, 54, 76, 100, 153

EOA Externally Owned Account. 37, 39–42, 55, 74, 76, 77, 99, 118, 120, 131

ESCC Endorsement System Chaincode. 64

EVM Ethereum Virtual Machine. 17, 43, 79, 139

GDPR General Data Protection Regulation. 6, 12

HTTP Hypertext Transfer Protocol. 16, 30, 43–45, 59, 76

IBFT Istanbul Byzantine Fault Tolerant. 100–110, 121, 126, 148, 149, 152

IDE Integrated Development Environment. 151

IoT Internet of Things. 7, 81, 84, 130, 147, 150

JSON JavaScript Object Notation. 28, 30, 35, 43, 59, 72, 76, 78, 112, 147

JWT JSON Web Tokens. 44, 48, 55, 76, 152

MSP Membership Service Provider. 24, 25, 55, 64, 70

MVCC Multiversion Concurrency Control. 70, 82, 83, 131, 133, 139, 141, 143

NFT Non-Fungible Token. 131

OSN Ordering Service Node. 22–24, 26–28, 51, 66, 67, 69, 81, 84, 88–90, 96–98, 105, 114, 135, 147, 148, 150

PDC Private Data Collection. 111–116, 125–127, 149, 153, 154

PKI	Public Key Infrastructure. 22, 24, 37, 132, 164
PoA	Proof of Authority. 99, 100
PoW	Proof of Work. 98–100, 105, 121, 148
RLP	Recursive Length Prefix. 76, 100
RPC	Remote Procedure Call. 16, 35, 43, 59, 72, 76, 78
RSA	Rivest-Shamir-Adleman. 37, 48
RSM	Replicated State Machines. 91, 92, 108
SCA	Smart Contract Account. 37, 39–42, 74, 116, 118, 125
SDK	Software Development Kit. 32, 59, 61, 63, 65, 66, 72, 81
TCP	Transmission Control Protocol. 78
TLS	Transport Layer Security. 25, 122
TPS	Transactions Per Second. 15, 18, 19, 132–138, 140, 141, 143, 150, 155
UI	User Interface. 13
URL	Uniform Resource Locator. 48
UTXO	Unspent Transaction Output. 9
UX	User Experience. 13
VIN	Vehicle Identification Number. 28
VM	Virtual Machine. 55
VSCC	Validation System Chaincode. 70, 82, 125

Chapter 1

Introduction

1.1 Blockchain to overcome traditional limitations

A new field of research emerged in 2008 after a (group of) individual(s) published a whitepaper under the pseudonym of Satoshi Nakamoto [68]. This paper highlights that traditional financial institutions are inherently flawed due to trust. To mitigate this, the paper proposes a new electronic payment system that relies on cryptography, called *Bitcoin*. This system's state can always be derived and verified thanks to its underlying mechanism commonly referred to as *blockchain*. A blockchain is a type of *ledger* in which the state of the system is made up of transactions, distributed over several participants. Using cryptography and peer-to-peer protocols, blockchain technologies aim to prevent any individual from making fraudulent changes, according to a set of predefined rules, to the ledger without agreement (i.e., *consensus*) of the network. Different flavours of blockchains exist, including *permissionless* aka *public* blockchains. These are blockchain technologies in which anyone can join the network, participate, and have the same permissions as others. Bitcoin is one such permissionless framework.

Collaborating businesses and organizations can face the same issues of trust. Consider a group of competitors that are mutually distrusting by nature. To make this more concrete, let us consider two banks in different countries. Sometimes, these competitors wish to collaborate for both parties' benefits. For instance, the two banks may wish to trade (or convert) currencies for a fair rate. Coming to an agreement on the details of the transaction, i.e. conversion rate, would take too long using paper-based contracts requiring mutual signatures. In fact, the global conversion rate may have already changed a lot in favour of one of both parties. Instead, automated digital systems should be used. Yet, neither party wants to rely on the other party's system, as this system may be designed to gain an edge, i.e. charge a higher conversion rate due to the time at which the transaction is carried out. Both competitors would benefit from using a tamperproof system in which they are able to transact in an automated fashion while ensuring all transactions with mutual agreement are recorded reliably. *Permissioned* blockchains are another flavour of blockchain aiming to solve the aforementioned problem.

Permissioned blockchains are, as opposed to permissionless blockchains, networks in which access is restricted [3, 74]. All participants are known and identifiable. Additionally and depending on the framework, different participants may have different roles and permissions, allowing them to perform different operations. These additional characteristics make permissioned blockchains very suitable for enterprise use cases involving collaborations between different organizations.

Besides the mitigation of requiring trust, the use of blockchain in an enterprise setting aims to provide other advantages. When multiple organizations are collaborating, each organization can host their own data if they wish to do so [3]. Depending on the permissioned blockchain implementation, ledger state may only be visible to a subset of participants [14]. Moreover, private transaction details may only be disclosed to the organizations involved in the transaction [4], if this is supported by the framework. Without permissioned blockchains, automated and contractual collaborations would be much more difficult to achieve [88]. Furthermore, the automation of sharing confidential data between organizations can save time and, thus, money.

1.2 Today's relevance of permissioned blockchains

Permissioned blockchains have proven to become more and more relevant today. According to the International Data Corporation, traditional business processes often involve manual, slow, and inefficient labour, which the COVID-19 pandemic has intensified [88]. As a result, the IDC reported that worldwide blockchain spending was forecast at 4.1 billion USD in 2020 alone. This corresponds to an increase of more than 50% compared to the year before [87]. Migrating legacy electronic systems or paper-based trails to permissioned blockchains could help these businesses improve existing processes, or help create entirely new use cases.

Coming back to the example of identity ownership, Verified.me[1] is a Canadian service giving citizens control over their identity. Citizens can share only relevant parts of their identity with financial institutions such as banks and insurance companies by using an app instantly, as opposed to labour-intensive manual work. They see exactly what data is being shared with whom. In 2018, the Netherlands Vehicle Authority (RDW) reported that the odometer of 21,6% of all imported second-hand vehicles from Germany are questionable [77], of which it has been established that at least half of these odometers were surely tampered with. Vinturas[2] is a consortium for achieving supply chain visibility for finished vehicle logistics in the European Union. By moving vehicle information, such as damages and odometer status, to the blockchain, the car's history at certain points in time will become transparent and can help to combat fraud. According to Vazirani et al., permissioned blockchains are currently being implemented in healthcare systems [94]. For instance, Guardtime[3] is one of the organizations on the forefront, as health records of millions of Estonians are already recorded on the blockchain.

1.3 Research Objectives

Most of these practical business cases are using underlying frameworks, on which their applications are created. Two such frameworks are Hyperledger Fabric[4] [3] and Hyperledger Besu[5]. From now on, they will be referred to as Fabric and Besu, respectively. While Fabric was designed with permissioned networks in mind from the start, Besu is based on Ethereum [97] and was adapted for permissioned systems. The goal of this thesis is to perform a deep technical analysis of both frameworks on a qualitative and quantitative level in order to answer the research question: "***What are the differences between Hyperledger Fabric and Hyperledger Besu, as well as their advantages and disadvantages?***"

The main research question is answered by comparing and evaluating both permissioned frameworks in the following manner. Each permissioned framework is individually looked at through a set of lenses. Then, after having discussed both frameworks through the same lens, they are contrasted to highlight their fundamental differences and the implications of these differences. The five different lenses through which Fabric and Besu are compared, are: network architecture, transaction flow, consensus protocols, data privacy, and performance. All chapters except on data privacy are written from the perspective that networks are private, but the transactions within that network are public. The chapter on data privacy then describes the changes required to the existing architecture and transaction flow, to highlight how the bases of both frameworks are adapted to support private transactions. Finally, multiple benchmark experiments are conducted on both frameworks by tweaking various variables identified throughout the analysis. All information on the frameworks is based upon the latest release of Fabric and Besu at the time of writing, corresponding to v2.2[6] and 20.10.4[7] respectively.

[1]https://verified.me/
[2]https://www.vinturas.com/
[3]https://guardtime.com/
[4]https://www.hyperledger.org/use/fabric
[5]https://www.hyperledger.org/use/besu
[6]https://hyperledger-fabric.readthedocs.io/en/release-2.2/
[7]https://besu.hyperledger.org/en/20.10.4/

First of all four lenses, the frameworks' architectures will be inspected from a top-down view in Chapter 3 to answer the following sub-question: "***How do the overall architectures of Fabric and Besu differ?***" Naturally, both blockchain frameworks are expected to be very different: Besu is built around the Ethereum Virtual Machine, whereas Fabric is designed without being restricted by a specification. What building blocks are required to deploy a network? Are the network topologies very different?

Secondly, the transaction flow of both frameworks is analyzed to answer the sub-question in Chapter 4: "***What is the transaction flow of Besu and Fabric from commit to validation?***" The text follows the same order from when a transaction was created, until the final commit to the ledger. In addition, this chapter concerns itself with the following questions in general: Who can create transactions? How are these transactions being created? How are these transactions distributed to the rest of the network?

Thirdly and crucial to the transaction flow, consensus protocols are required to agree on how blocks are created. As we will see in Section 2.3.1 and 2.3.2, Fabric and Besu both support modular consensus. Therefore, Chapter 5 answers the sub-question: "***How do the modular consensus protocols work, and what are their differences?***". To find out these differences, this chapter focuses on the technical details of how each individual consensus protocol works.

The fourth lens regards data privacy. In an ideal situation, no data is shared with any participant, unless they themselves are involved in the transaction. However, this is often difficult when considering ledger integrity. Thus, Chapter 6 aims to answer the sub-question: "***How is data privacy achieved in both frameworks, how is this information stored, and what does this imply for data availability and integrity?***"

The fifth and final lens concerns itself with the performance of both frameworks in practice in Chapter 7. "***How does the performance of both frameworks differ in terms of throughput, latency and resource consumption, when considering various framework configurations?***"

1.4 Contributions

The contributions of this thesis are manifold. This thesis provides an in-depth technical analysis on both frameworks by not only looking at what functionality they support, but by investigating how they are implemented and what implications they have. As such, Fabric's technical analysis often goes beyond the information provided in its documentation and can be considered the first contribution. As Besu's documentation is very limited and no papers have been published on its architecture at the time of writing, information on Besu can be regarded as a contribution by itself. Thirdly, both frameworks are compared for each of the aforementioned five lenses, giving the reader insights in how (and whether) each framework solves a particular problem. This information gives competitive insights in both frameworks. Fourth, this thesis not only investigates the features and pros of each framework, but takes a critical viewpoint as well by highlighting their flaws and shortcomings. Fifth, benchmarks give the reader an intuition of how both frameworks perform side-by-side, using similar network configurations. Sixth, several open issues including bugs, inefficiencies, flaws and future directions are provided for both blockchain frameworks and associated projects. GitHub issues in the respective project repositories for (most of these) open issues are listed in Appendix B and were open at the time of writing. The resources used during the work of this thesis (including source code, scripts, and figures) can be found on GitHub[8].

1.5 Outline

The remainder of this work is structured as follows. Chapter 2 provides background information on (permissioned) blockchains, their challenges, existing frameworks and related work. Next, Fabric and Besu are analyzed from different perspectives: Network architectures in Chapter 3, Transaction Flow in Chapter 4, the underlying Consensus Protocols in

[8]https://github.com/Viserius/FabricBesuBenchmark

Chapter 5, and Data Privacy in Chapter 6. Chapter 7 discusses various experiments on benchmarking both frameworks. Then, the work of this thesis is reflected upon in Chapter 8 and is concluded in Chapter 9.

Chapter 2

Background Information

The term "blockchain" is commonly used within the context of Distributed Ledger Technologies (DLTs). Section 2.1 explains what DLTs are, and elaborates on the most noteworthy technology concepts, including blockchain. Then, the scope is narrowed down to permissioned blockchains in Section 2.2, by highlighting the differences compared to permissionless blockchains. Consecutively, some of the more prominent challenges that arise with current permissioned blockchain frameworks, as well as using them to design a network, are discussed in Section 2.2.2. An overview is given of the state of the art permissioned blockchain frameworks in Section 2.3. Lastly, Section 2.4 discusses work related to this thesis.

2.1 Distributed Ledger Technologies

A Distributed Ledger Technology can be defined as a database of transactions that is distributed over multiple participants or institutions [95]. This database does not describe the system's *state*, but these transactions describe *how* the state was *modified*. This database of transactions, is also known as *ledger*. All participants in the network usually have access to this ledger at any point in time, and if they wish, may maintain their own copy of this ledger. Although many traditional database systems only maintain their state at the latest point in time, a DLT's state is derived by applying the changes described in all historic transactions. Often, a ledger maintains a view of this state such that the ledger's entire state does not need to be derived from scratch for every read.

Different DLTs may support and record different types of transactions on the ledger. One type of transaction could allow users to transfer assets representing monetary value such as a *cryptocurrency*. Alternatively, users may be able to deploy or invoke a *Smart Contract* if this is supported by the DLT. A smart contract is a programmable set of business procedures, typically agreed upon directly or indirectly by its users. When a user creates a transaction to invoke a smart contract, (a subset of) the network participants execute the referenced business procedure. The results of executing this business procedure are then written to the ledger's state. Sometimes, smart contracts are composable, meaning smart contracts may invoke each other. Essentially, smart contracts are the means for developers to program and deploy applications in a decentralized and distributed manner on the DLT.

Since DLTs are distributed by definition and trust between network participants is (assumed to be) absent, a distributed *consensus protocol* is required to ensure there is network-wide agreement on the state of the ledger. The consensus protocol usually results in agreement on the *ordering* of transactions. Sometimes, the consensus protocol also ensures that ordered transactions must be *valid*, based on a set of predefined rules. We speak of *consensus* if there is network-wide agreement among the participants on the (state of the) ledger as described by its transaction ordering, and optionally, transaction validity.

One of the main goals of DLTs is to allow participants to interact with each other in a trusted manner, without the need of a trusted third party [68, 95]. Most DLTs achieve this goal by pursuing four characteristics: transparency, traceability, security and fault-tolerance. Transparency and traceability are required such that one is always able to recompute the current ledger state from all historic transactions. This allows participants to ensure that the agreed-upon state is "correct", in terms of rules defined by the network. Most DLTs allow participants to track transactions (and digital assets) over time, by consulting the transaction history. Security includes resilience to eliminate the need for trust, by ensuring it is (nearly) impossible to tamper with the data on the ledger. Fault-tolerance is key to ensure that the network can continue operating while a part of the network is unavailable. These quality attributes are achieved through cryptographic primitives, peer-to-peer networking protocols, consensus mechanisms, and sometimes even economic incentives [31, 39].

Different types of DLTs exist that use different data structures under the hood, such as blockchain [68, 97], tangle [75], hashgraph [9, 8] and sidechain [84]. One should be aware that these are only high-level *concepts*, as opposed to concrete *implementations*. Individual implementations may prioritize some (different) quality attributes at the cost of others. As an example, some (enterprise) implementations may find privacy of greater priority, by sharing transaction details with as few participants as possible. Alternatively, other (public) implementations could prefer security and ledger integrity, by requiring all transaction details to be shared with (and executed by) all participants. Sometimes persistency is crucial, in the sense that a committed transaction is irreversible, while in other cases, data should sometimes be removed to comply with regulations as the General Data Protection Regulation (GDPR). At least 22 different consensus protocols have been analyzed for various properties by Xiao et al. [98]. Some DLTs only support a single consensus protocol, while others may support multiple. In any case, this highlights the wide variety of DLT implementations and how they can be configured in terms of consensus.

Later in this work, (the technicalities of) Fabric and Besu are covered in great detail. Therefore, the remainder of the background section does not focus on concrete *implementations* of DLT frameworks. Instead, a brief overview is provided of the various DLT *concepts*, including blockchain, tangle, hashgraph and sidechain. Although *DLT* is an umbrella term for the technological concepts, and the term blockchain refers to a specific subset of DLTs, both terms are commonly used interchangeably in the literature. Therefore, after this section, both terms may occasionally be used as such.

Blockchain

In a typical blockchain setting, the ledger consists of a linked list of blocks. These blocks are interconnected, in the sense that each new block contains a hash linking to the previous (parent) block in the linked list. Consequently, if any block in the chain is invalid, all consecutive blocks will be invalid as well, since they are the child of a dishonest parent block. A timestamp is usually included in the block, as proof that the block existed at some point in time it was created [68]. A visual representation is shown in Figure 2.1.

A blockchain participant who wants to create a transaction cannot simply append this transaction to one of the existing blocks. Transactions are generally only added to new blocks. The participant is usually required to create the transaction him- or herself, and share this transaction with (a part of) the network. To prove the transaction was created by some individual, the creator has to sign the transaction. The consensus algorithm determines which transactions are added to the blocks, which transactions are rejected, in what order transactions are executed, and what nodes are allowed to form these blocks. Then, one or a small group of nodes will form the blocks and distribute these new blocks with the rest of the network. Since the consensus algorithm is highly specific per implementation, exact details are not further discussed.

Tangle

Instead of using a linked list in which the nodes are blocks, a tangle uses a Directed Acyclic Graph (DAG) as the underlying data structure. The tangle was originally designed by the Iota Foundation[1] in order to solve some of the limitations and

[1]https://www.iota.org/foundation/vision-and-mission

drawbacks of the blockchain [75]. Initially, they argued traditional blockchains require transaction fees as an economic incentive for creating blocks. According to [75], having transaction fees make blockchain infeasible for Internet of Things (IoT) use cases. They also highlighted the fact that different types of participants (block creators and users) may lead to conflicts of interest.

Instead of grouping transactions into blocks that form the nodes of the linked list (like in a blockchain), in a tangle, individual transactions form the nodes: blocks do not exist. Edges represent transaction validations. For a user to append a new transaction to the graph, the user must validate a certain amount of prior transactions, e.g. in Iota 2 validations are required for each new transaction. Because of this, transaction fees are obsolete as users are performing (most of the) validations themselves. A figure of the tangle is given in Figure 2.2.

The tangle was initially deployed on the Iota permissionless blockchain[2]. Their initial release (version 1.0) still required centralized coordinators to create milestones and snapshots of the blockchain for reliable consensus purposes, which severely limited their transaction throughput to 20 transactions per second at most and validation time of longer than 80 seconds. After a new release eliminated the need for these centralized coordinators, the tangle has been shown to support 1500 transactions per second with validation times of about 10 seconds[3]. Their paper on an upcoming release claims the tangle can achieve a theoretical infinite throughput, due to the elimination of centralized coordinators and the entire network having to validate transactions [76].

So far, a tangle has only supported financial transactions and not smart contracts. Smart contracts have been in development, although these are executed by a layer-2 "Off-Tangle" network on top of the underlying tangle, introducing separate *committee* nodes and transaction fees[4]. Since they deviated from their original goal to have no fees and uniform nodes, I wondered if this was due to some limitations of the tangle. Perhaps, a tangle would not be fit for smart contract execution, and thus, not suitable for enterprise blockchain use. Therefore, the author of this thesis reached out to the developers at the

[2] https://iotatoken.com/
[3] https://blog.iota.org/chrysalis-iota-1-5-phase-1-now-live-on-mainnet-958ec4a4a415/
[4] https://blog.iota.org/an-introduction-to-iota-smart-contracts-16ea6f247936/

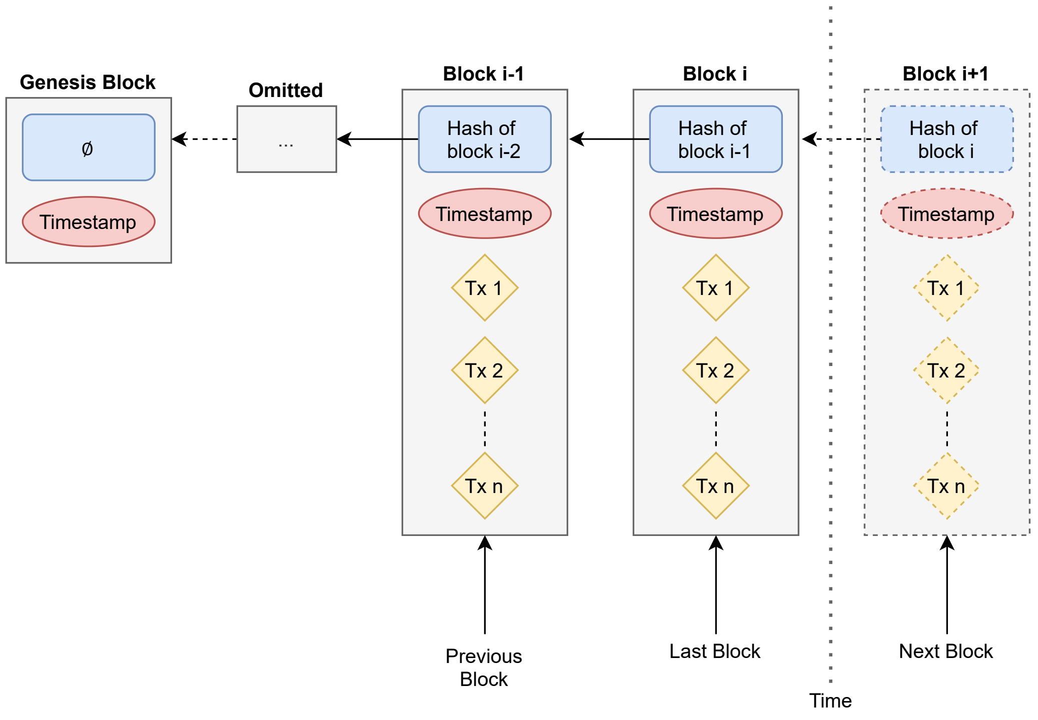


Fig. 2.1 Blockchain DLT

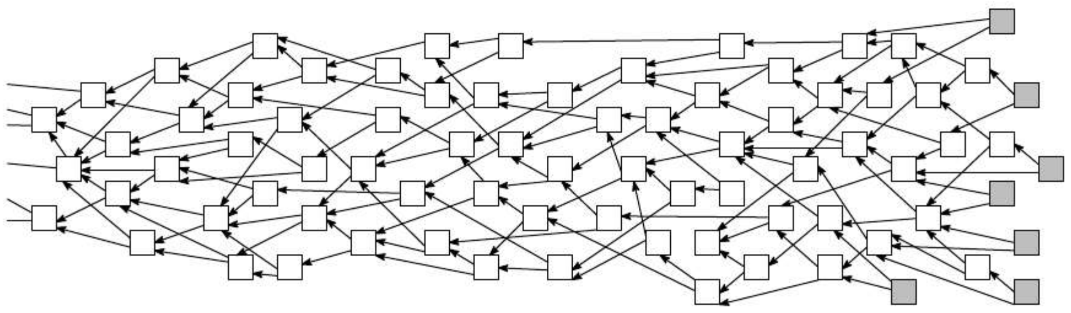


Fig. 2.2 Tangle DLT [75]

Iota Foundation about this design decision on one of their main communication channels and asked if they could have implemented smart contracts using the original tangle flow as well[5]. A developer replied that they aimed to have smart contracts that could interact with the global state of the system. Within the tangle, there are many streams of transactions added in parallel and the "DAG has no consensus on the total order, as the total order is only possible eventually, after a long time"[6]. The solution they are working on, is to have a more traditional blockchain in parallel for each smart contract, that only needs to be verified by a subset of nodes, as opposed to the whole network. This developer mentioned, "each one such chain is not scalable, there's a consistent local state on it, however, we can run many of them in parallel".

Although still experimental, this project proposes an alternative DLT to the widely used concept of using blocks in a blockchain, which could prove useful for optimizing the *scalability* of other existing and new projects. However, it should be noted that the concept of a tangle has not been demonstrated on a large scale (yet).

Hashgraph

A hashgraph is another type of DLT proposed by Baird [7] and is implemented in the Hedera permissioned blockchain network [9]. A user submits the transaction to a *trusted* peer. A *gossip* protocol is used between trusted peers to disseminate the transactions. A hashgraph is similar to the tangle in the sense a DAG is used as the underlying data structure. Unlike a tangle, however, nodes in the DAG represent transactions as well as hashes of previous gossip events. As such, the gossip protocol is not used to gossip just the *transaction* itself to other peers, but also to gossip the last received gossip events to other peers. For instance, Alice's node may send the following information to Bob's node: *"I received* TRANSACTIONS[] *at* TIMESTAMP*; The last event I, Alice, created is* HASH1*; The last event I received from Bob is* HASH2*;"*. A representation of such a hashgraph is given in Figure 2.3. Each edge represents sharing gossip information, whereas each column represents a single peer. Additionally, every single transaction must be executed and verified by all trusted peers to be valid.

Despite the communication overhead of gossiping additional hashes instead of the sole transaction and the fact that all nodes must execute the transaction, Hedera's implementation claims to be capable of "thousands of transactions per second in a single shard" due to new transactions spreading "exponentially fast through the community until every member is aware of it" thanks to the gossip protocol [9]. This high throughput applies to the network as a whole, i.e. the throughput of a single smart contract can be much lower, as becomes clear in the next paragraph. Through gossiping, *fairness* in terms of the ordering can be guaranteed. Whether transactions invoke a smart contract or transfer financial assets (e.g. cryptocurrency), they will always be added to the ledger based on the earliest timestamp.

Since Hedera is a fork of Ethereum, smart contracts written in Solidity are supported. Despite Hedera's claims to support thousands of transactions per second, tune.fm[7] reported that deploying their code as a single smart contract was deliberately throttled to 10 transactions per second, to achieve fast consensus time [5]. Due to tune.fm's fast growth, this throughput

[5]https://discord.com/channels/397872799483428865/441598975066374145/807305046781329448
[6]https://discord.com/channels/397872799483428865/441598975066374145/807335897736806451
[7]https://tune.fm/

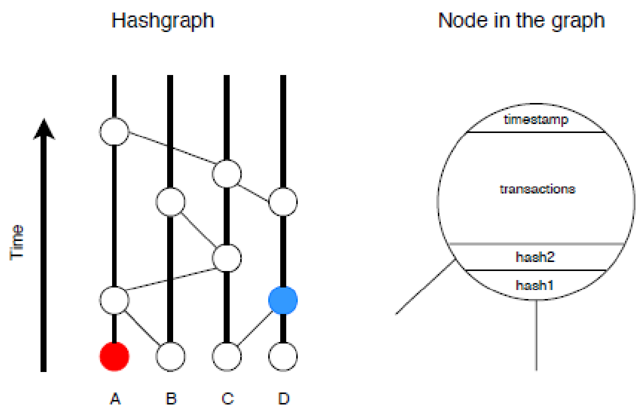


Fig. 2.3 Hashgraph DLT [31]

quickly became a bottleneck. Their solution was to break up their code into multiple smart contracts, which they called "smart contract sharding". This would distribute all transactions over multiple contracts, meaning consensus can be reached in parallel (since they are essentially deployed in different namespaces) to improve throughput.

To summarize, it should be repeated this information applies to one implementation of the Hashgraph DLT, it illustrates the same struggle the tangle was having: deploying smart contracts on a DAG data structure results in a trade-off between high throughput and (fast) consensus on the transactions. To summarize, there is a discrepancy between the high throughput of transferring a cryptocurrency and the low throughput of executing a smart contract. This is caused by the fact that smart contracts are designed to invoke any other smart contract in the network and therefore require a complete and stable view of the entire ledger's world state. On the other hand, a financial transaction only relies on a stable view of an account's balance (as is the case with the account-based Hedera/IOTA), or a stable view on the coins before they are spent (Unspent Transaction Output (UTXO)).

Sidechain

All DLTs discussed in this paper so far, have the requirement that there needs to be consensus by the entire network on the ordering and existence of smart contract transactions at certain points in time, also known as *finality*. Especially in enterprise use cases in which the participants are known, this requirement could be relaxed by only requiring consensus between subsets of participants.

A paper by Back et al. introduces the concept of sidechains [6]. The paper introduces a *two-way peg* to allow the transfer of (cryptocurrency) assets between different sidechains, to overcome performance limitations of the main network. Robinson et al. expand on this by proposing a technique that allows sidechains to be used in conjunction with smart contracts [80]. Instead of having one large network, this network could be split into many smaller subnetworks, which are essentially private DLT networks. These sidechains have their own DLT implementation, mechanisms, and consensus protocol. Since sidechains can hide the identities of participants for the rest of the network, as well as restrict who can connect to their sidechain and access their data, they are ideal for business use cases. Robinson et al. write that sidechains could be used for groups of business partners, that want to transact privately. To transact with other groups, a coordination DLT is used as an intermediary. Confidential data such as contract details can be omitted while interacting with the coordination DLT. Then, these data are available to other sidechains with groups of partners that can read this information from the coordination DLT. This concept is illustrated in Figure 2.4.

According to Robinson, this technique would work on many different blockchain systems, as long as (1) it is possible to query information from other blockchains, and (2) functions of other blockchains can be invoked (through mechanisms like RPC for instance). Robinson weights the advantages and disadvantages of these private sidechains on the Ethereum

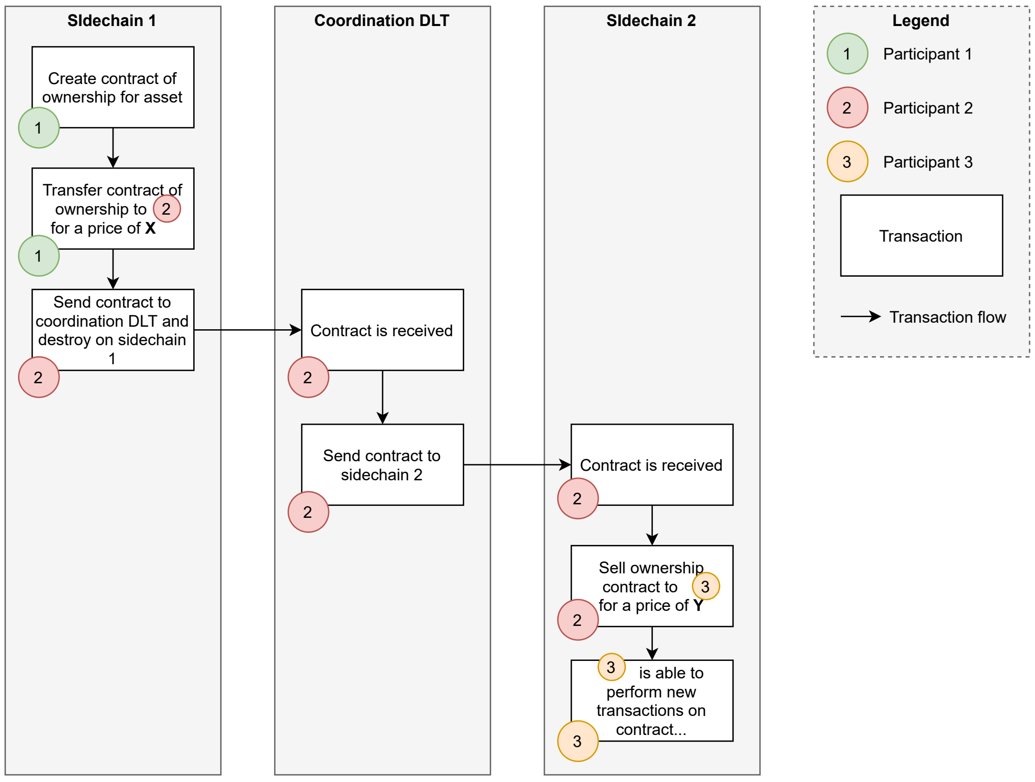

SIdechain 1
Create contract of ownership for asset
1
Transfer contract of ownership to 2 for a price of X
1
Send contract to coordination DLT and destroy on sidechain 1
2
Coordination DLT
Contract is received
2
Send contract to sidechain 2
2
SIdechain 2
Contract is received
2
Sell ownership contract to 3 for a price of Y
2
3 is able to perform new transactions on contract...
3
Legend
1 Participant 1
2 Participant 2
3 Participant 3
Transaction
Transaction flow

Fig. 2.4 Concept of Sidechain

network in another paper [79]. Furthermore, this concept of having one main DLT that is linked to other separate DLTs in parallel is similar to how Iota is planning to add smart contracts to the tangle. The main difference, however, is that they will store all transactions on the parallel DLTs, while sidechains will post this shared information on the main coordination blockchain. Nevertheless, there is a reoccurring pattern of having the main blockchain network linked with possibly private DLTs to keep the global state as simple as possible for performance considerations.

2.2 Permissioned Blockchains

Permissionless blockchains, DLT networks in which anyone can join and participate in the network, suffer from a handful of limitations according to Hamida et al. [39]. As a result, the industry started to develop their proprietary DLT platforms suitable for enterprise use cases. Not only are these platforms often developed by businesses, but they are often operated and maintained by businesses, rather than individuals. Different access control policies and authorizations may be in place: different participants will have different roles, and thus, can perform different operations. DLTs with these characteristics are often referred to as *permissioned blockchains*. More formally, a permissioned blockchain network is defined as a network in which access of identifiable participants is restricted [3, 74]. The Chartered Professional Accountants of Canada and the American Institute of CPAs acknowledged permissioned blockchains have great potential to maintain privacy and fit business, whereas permissionless blockchains do not [15].

The observant reader may have noticed that the terms "permissionless blockchains" and "permissioned blockchains" are used to refer to business DLTs in general, rather than specifically referring to the blockchain DLT. Indeed, the terms *permissionless blockchains* and *permissioned blockchains* are not by definition bound to a technology involving blocks. They may also concern other types of DLTs. However, to stick to the widely accepted terminology within the literature, the remainder of this thesis will use both terms to refer to DLTs in general, whether their implementation actually uses blocks or not.

2.2.1 Overcoming limitations of permissionless blockchains

According to Hamida et al., there are five limitations many permissionless blockchains suffer from, that permissioned blockchains aim to overcome [39]. This subsection summarizes these limitations and briefly explains how these limitations are mitigated or even eliminated in permissioned blockchains.

Irreversibility of data

Data committed to a ledger cannot be reversed in traditional permissionless blockchains. The data stored on permissioned blockchains are often subject to changes in the real world. As an example, a contract that was signed by two parties may be declared void because of a violation or a judge ruling. Or, perhaps, because the two parties agreed to reverse the contract outside of the blockchain. A permissioned blockchain can relax the inherent irreversibility of permissionless blockchains, by allowing participants to undo a transaction under strict conditions.

Public Ledger

Most permissionless blockchains publish all transactions and data publicly. Since anonymous participants are distrusting by nature, any participant should be able to access all transactions and information in the ledger state. Businesses do not want their confidential information publicly available, since they may contain secrets that can give competitors an advantage. As a first step, by having access to the network restricted only to identified individuals, data privacy is already achieved for non-participants. Some permissioned frameworks take this one step further, by only exposing the contents of individual transactions to a subset of the network.

Limited scalability due to security

As mentioned before, anonymous participants may want to make illegal transactions for their own benefit in permissionless blockchains. As a result, permissionless blockchains require very robust consensus protocols that can severely limit the throughput of the network. As Gupta et al. conclude: "Key components of *any* permissionless and permissioned blockchain remains the underlying Byzantine Fault Tolerance (BFT) consensus protocol [...]. These protocols are challenged by the scalability and performance required [...]." [38]. Gupta et al. further write that a consensus protocol that is BFT, does not scale well, since introducing more nodes only decrease network throughput.

While distrust and the risk of illegal transactions are still present in permissioned blockchains, participants are identifiable thanks to access control to the network and digital signatures. Therefore, if an illegal transaction is noticed by the network, this may result in consequences in the physical world. Examples could be to remove the party from the network or to file a lawsuit. Consequently, permissioned blockchains may choose to allow for less security in favour of more scalability and performance. For instance, instead of requiring every node in the network to verify all transactions, it could be enough to have a transaction verified by a subset of nodes.

Low responsiveness due to high probability of forks

Permissionless blockchains may have a relatively high probability of a *fork* occurring in the blockchain due to many individuals trying to verify transactions (and create blocks) in parallel. When a fork occurs, the network needs to reach consensus on the ledger state, including which transactions are accepted in which order. Therefore, it may take several minutes until a consensus is reached and the transactions are committed to the ledger. The higher the probability of forks, the lower the latency of transaction validation. Permissioned blockchains can overcome this limitation by having different consensus protocols and validation mechanisms to reduce or eliminate forks. By reducing the probability of forks, transactions are validated faster.

Difficult to perform updates

Permissionless blockchain platforms may be difficult to update, especially if the new changes are breaking and not valid on the old version. Due to the difficulties of reaching consensus on accepting a new release under the anonymous participants, this sometimes results in a *hard fork*. This means the participants have to abandon to the old blockchain and switch over to a new permissionless blockchain instance based on the same ledger data. Often, this process is slow and difficult. Mutual agreement is easier to reach in a permissioned blockchain setting, since the participants know each other and discuss the changes together more easily. Sometimes, permissions are in place allowing a certain organization to perform these updates.

2.2.2 Challenges of permissioned blockchain frameworks

Although the previous section explains how many limitations can be overcome, there are still many challenges regarding permissioned blockchains. This section highlights some of these technical and non-technical challenges as identified by the literature. Permissioned blockchain frameworks should aim to resolve as much of these challenges as possible, although some of these non-technical challenges exist external to the blockchain implementation and may only be resolved per use case. Moreover, some challenges (such as security and privacy) contradict each other and result in a trade-off. Naturally, no *one-size-fits-all* framework exists.

Required Proficiency

As elaborated by [60, 45], technical expertise of blockchain technology is required to create, maintain or participate in blockchain networks. Especially for smaller firms with less technical knowledge and fewer employees, it becomes harder to participate. Consider the tracking of food from farm to consumer [51, 12]. In the case of food contamination, only contaminated food should be recalled while uncontaminated food should continue to be sold. Such a business case only works if local supermarkets and greengrocers participate. Especially for these smaller and local businesses, little *understanding* of the technology can become a barrier for successful adoption, requiring the right education. Of course, an expert in the field (like a consultancy company) could be hired to abstract the details for these local businesses and only educate them on the benefits the technology can bring them.

Moreover, *using* permissioned blockchain frameworks requires knowledge of these proprietary systems in order to configure, program, use and operate them [45]. Similarly to the previous paragraph, an external expert can be hired to perform these tasks on the company's behalf, but introduces additional costs, trust and reliabilities on an external party. While no standards have emerged yet, the barrier of entry can be substantially large [57]. Some of the approaches taken by various companies with permissioned blockchains are to (1) define their own standards; (2) adapt already established standards for blockchain; or (3) apply existing standards where possible [57].

Complying with Regulations

According to [57], regulations around the world are having issues adapting laws due to the novelty of blockchain. Frizzo et al. concluded that 34% of blockchain business papers reported regulation as a challenge [34]. This may either be due to a lack of regulations or due to regulations that are rapidly changing [45]. The GDPR dictates that users should have access to viewing the data companies have on them and how this information is processed, as well as the right to erase this data. This becomes challenging for permissioned blockchains when user information is involved that may be private and not easily accessible. Furthermore, in case the user requests deletion, mechanisms should be put in place to not just erase this information from the global ledger state, but also from the transaction history. Lacity concluded that some firms are simply adapting their blockchain solution to comply with current regulations, while others are actively lobbying to adapt existing regulations [57]. Helliar et al. give examples of how new laws in various regions of the world made it more difficult for blockchain use cases [45]. On the other hand, there have been laws to make enterprise blockchain adoption easier, such as a Washington State law to legally recognize electronic records on blockchains [45].

Difficult to Audit

For larger businesses, especially in the financial sector, audits have to be performed on a regular basis. Despite some efforts to legally recognize blockchain records, performing audits are still a widely recognized challenge [39, 60, 45] due to a variety of reasons. It may be difficult for auditors to understand and use DLTs. In some cases, auditors have to wait for parties to export and send data from the blockchain as evidence. In other cases, an auditor with explicit access to the ledger can retrieve this information by him- or herself. Either way, an auditor has to review the ledger state and (a subset of) transactions, which could be complicated if they are unfamiliar with the framework, data structures, and storage formats. If the auditor has to retrieve this information by accessing the ledger by him- or herself, this requires even more expertise of the auditor. A blockchain implementation or business consortium may support this process by providing User Interfaces (UIs) to enhance the auditor's User Experience (UX). Considering permissioned blockchains are tailored for businesses that may have to perform these audits, this challenge is of particular interest to permissioned blockchains.

Although consensus protocols aim to eliminate illegal data on the ledger, they only ensure ledger integrity to a certain extent. Many consensus protocols are not completely secure (e.g. eclipse attacks [16], 51% attack [83], timejacking attack [67])[8]. This complicates the auditing process even further.

Alternatively, the ledger data may not resemble the physical world due to various reasons, such as human error. Consider a vehicle's odometer that was accidentally entered with a decimal in the wrong place. Therefore, auditors will still have to investigate whether the data on the ledger are accurate.

Hence, designing auditable blockchain solutions is difficult due to a variety of reasons. Some of these reasons (such as the auditor's understanding of the data) can be mitigated by the framework. Other difficulties may be external to the framework and should be tackled per use case. Consider the context of a supply chain use case, for example. Data on the ledger may be auditable, but it could be difficult to prove a physical asset is the original asset described on the ledger, as opposed to a substitute. Efforts have been made to physically link goods with an asset on the blockchain through crypto anchors [10]. Thus, a use case requires proper design of data models and possibly additional solutions external to the framework to make the use case viable for performing audits.

Sufficient amount of Data Privacy

In Section 2.1, we described how different DLTs focus on different quality attributes. Achieving privacy is a very important quality attribute for permissioned blockchains. It is important to distinguish between *anonymity* and *data privacy*. *Anonymity* refers to the unidentifiability of participants. *Transaction anonymity* means that someone who reads a transaction is unable to establish the individual originally committing it. Similarly, *network anonymity* is another form of anonymity in which a participant does not know the identities of all other participants in the network. *Data privacy*, on the other hand, means that the actual *contents* of a transaction is only known by a subset of participants and secluded from others. The extent of privacy is flexible and comes in different flavours per framework and per use case: E.g., a transaction may only be shared with a small and select group of participants referenced by the transaction's contents, or a transaction may be shared with all participants in a private sub-network.

According to Hamida et al. [39], many techniques to improve privacy exist. Examples of these techniques are Zero-knowledge proofs, Homomorphic Encryption, Pedersen Commitments, Stealth Addresses, and Ring Signatures [39]. Hamida writes, however, that none of these aforementioned techniques hide the identities as well as the transaction contents at the same time. In practice, permissioned blockchain frameworks find ways to combine those techniques. Some

[8] Although attacks have mostly been studied on permissionless frameworks, permissioned frameworks are not without risks.

permissioned blockchain frameworks may only shield data from non-participants. Then, there is a risk that a participant with bad intent sells or leaks the information on the ledger [60].

Security and risk of malicious actions

Often, a trade-off has to be made between privacy and security, as explained next. The higher the degree of privacy, the less participants receive and read a transaction. In most cases, this also means that a transaction is validated by less participants. As this group becomes smaller, it becomes increasingly easier to tamper with the data as consensus protocols only provide security to some degree. On the other hand, the more participants that receive, read and validate transactions, the harder it becomes to exploit the consensus protocol. Therefore, creating illegal transactions or manipulating the ledger becomes increasingly more difficult with the number of validating participants required. This results in a direct trade-off: the more secure the blockchain, the more nodes will have to read and verify the transaction, and thus, the less privacy. This particularly applies to permissioned blockchains. In a private setting when a car is sold between a buyer and seller, while nobody else sees the transaction and asset, both parties may be able to tamper with the data and dial back the odometer.

Some permissioned blockchain implementations may only verify transactions by a subset of central (notary) nodes. While this ensures the transaction contents do not have to be verified (and thus, seen) by the rest of the network for high privacy, it may lead to centralization [44]. As a result, such a central authority may be able to abuse its power [60]. For instance, this authority may decline transactions of a specific participant (possibly a competitor), or collaborate with one of the participants to tamper with the data.

Hamida et al. [39] write that any DLT business model that does not aim to remove or reduce trust on any party would eventually result in failure, as a much simpler centralized system could be used. Additionally, Hamida writes that members of a permissioned blockchain should all have financial interests in participating, albeit through receiving revenue or reducing costs. By benefiting from the collaboration and suffering when not participating, members would be less inclined to perform malicious actions.

Shared Governance

Another challenge related to managing permissioned blockchains concerns governance. How is one able to change consensus protocols? How does one decide which new features are added, and which are not? How does one determine if a new participant is allowed to enter the network? Who pays for the operations, server costs, and maintenance?

Hamida et al. identify two different types of permissioned blockchains [39]. First, there are *private permissioned blockchains*, which are managed by a single organization. Second, there are *consortium permissioned blockchains*, which are managed by a group of participants. Obviously, the first is easy to manage, since a single organization can make important decisions all by itself. The latter, however, requires coordination and agreement as a group. Such a formal structure in which decision making is performed by multiple (possibly) distrusting consortium participants is referred to as *Decentralized Governance*.

Two examples of popular decentralized governance structures commonly used in permissioned blockchains are *majority voting* and *unanimous voting*, explained next. As the name suggests, majority voting could be used by consortia to only accept a proposal once a majority is in favour of the proposal. However, this may put some participants in a disadvantage. Consider, for instance, when a direct competitor of another participant is added by majority vote. This results in a tradeoff: growth in the number of participants usually benefits the adoption, but too many participants may result in smaller financial gains of individual participants [39]. Secondly, unanimous voting may be used in which a proposal is only accepted once every single participant agrees with it. This decentralized governance model ensures no participant is put at a disadvantage by a decision it did not support. Unanimous voting may work better in smaller consortia than in larger consortia. Namely, as the number of participants grow, the network becomes increasingly more rigid as fewer changes are likely to be approved.

Lacity et al. look at how some example consortium permissioned blockchains handle governance [57]. A privately owned startup, that builds peer-to-peer markets for energy trading, introduces a nonprofit organization to listen to the interests of (smaller) participants. A traditional enterprise for manufacturing aims to use a permissioned blockchain to track the authenticity of goods. This enterprise decided to create separate joint ventures and blockchain consortia per industry: a healthcare consortium, an aerospace consortium, et cetera.

Performance and Scalability

Permissioned blockchain frameworks can generally achieve higher performance than permissionless blockchains, as explained in Section 2.2.1. Typical performance measures for DLTs are transaction throughput in terms of Transactions Per Second (TPS), latency until a transaction is validated, and resource utilization such as memory, Central Processing Unit (CPU) and storage.

Within the context of blockchain networks, scalability is commonly denoted as the degree to which the throughput increases, as the number of computational power (i.e. nodes) are added to the network. However, consensus protocols often require each transaction to be validated by more than one node, resulting in poor scalability. In practice, adding more nodes to the network might even decrease the throughput[9]. If the throughput decreases when more nodes join the network, the delta of the throughput becomes negative and the system is said to be not scalable. In an ideal setting, the throughput should grow linearly as nodes are added to the network.

Scalability is still a challenge [45] that different permissioned blockchains try to solve in different ways. As a consequence, there may be large differences in raw throughput and scalability per framework. A common pattern we have seen in Section 2.1 with the tangle, the hashgraph and sidechains, is to divide the network into many smaller ledgers, rather than a single large ledger. Sharding is another technique that uses the same idea of subdividing the ledger [61]. Additional techniques used to achieve scalability could be to create a hierarchy of blockchains [39].

Adoption into existing business models

Enterprise business processes can be large, rigid, old and difficult to modify. That can make it difficult to adapt a pre-existing business model by shifting from a traditional database to a blockchain technology [65]. This is especially true for older and more rigid enterprises in which individuals are used to conducting business in a certain way using their priorly established set of (business) procedures [93]. Additionally, procedures for performing structural changes to data models may also be deeply embodied in the organization's processes, such as for migrating data, adapting data structures et cetera. Changing an established enterprise's business model for incorporating a fundamentally novel technology requires a paradigm shift [39]. In the worst case, shifting to a permissioned blockchain framework requires restructuring on the organizational level [11] (e.g. when the old database system is the primary responsibility of an entire team or department). Although this drawback applies to all new and groundbreaking technologies, it could be a serious barrier for adoption.

Integrating into legacy systems

Very similar to adoption, technically integrating a novel blockchain platform with traditional enterprise systems may form a barrier [53]. Legacy systems may comprise hundreds if not thousands of interconnected applications. It should be pointed out that a data store is usually located at the lowest layer or tier of any architecture. Moreover, legacy applications are often built directly on top of this data layer without re-using components. Switching out a traditional database system that has been used for tens of years for a completely different blockchain system may require many of these applications to be changed, resulting in a slow and expensive migration process [28]. It definitely does not mean it is impossible to perform these changes, just that it may be a serious obstacle for large enterprises that have been around for decades.

[9]This could be the case when every single participant must validate every single transaction, as the time to reach consensus increases linearly.

Interoperability by and with other systems

Since a permissioned blockchain forms the data layer, the permissioned framework should make it easy for external business logic to interact with it. Of course, different permissioned frameworks expose functionality in a different manner over different communication protocols. For instance, some frameworks might expose Application Programming Interface (API) functionality over Hypertext Transfer Protocol (HTTP), whereas others might communicate using a Remote Procedure Call (RPC) using a format specific to a certain programming language. To make this more concrete, consider a permissioned framework that reads incoming requests by using a Jakarta Message queue for Java or a message queue using standards of the Windows Communication Foundation for C#. Then, it could be difficult for systems written in C or Python to communicate with the permissioned blockchain framework. To conclude, the way that a permissioned blockchain framework exposes functionality could either make it easier or more difficult to be invoked by the existing systems. For the most flexibility, a permissioned blockchain framework should expose functionality independent of any other platform or programming language.

Another form of interoperability is the communication and coordination between different blockchain networks and frameworks. Consider the following scenario. A handful of businesses form a consortium for managing the logistics and traceability of parcel delivery within a country. Another country may have used a different permissioned framework for handling parcel delivery within their country. Or, we could have another permissioned network for parcel delivery overseas. At a later point in time, any of these three blockchains might wish to expand by incorporating functionality with another blockchain. Interoperability between different blockchain networks or frameworks is still challenging and a field of ongoing research [13].

2.3 Existing Permissioned Blockchain Frameworks and Consortia

Around 2014, corporations started investigating using permissioned blockchains for their businesses. R3 Corda[10] was founded in 2014 by some of the world's largest banks to use permissioned blockchain solutions[11]. After concluding that no framework existed to satisfy their requirements, they started developing their own permissioned blockchain. Pilots were still at an early stage in 2016 [45]. In 2019, they launched a transaction system in collaboration with over 50 banks across 27 countries called the "Voltron Letter of Credit" [49]. In 2020, the Italian Banking Association's banking blockchain network included about 100 Italian banks, or 18 banking groups [40] using R3 Corda[12].

Around 2016, JPMorgan Chase also started developing their own permissioned blockchain as a fork of Ethereum called Quorum [47]. In 2017, JPMorgan used this platform to create a peer-to-peer data transfer network for financial institutions called the Interbank Information Network [62]. While ConsenSys[13] acquired JPMorgan's Quorum in August of 2020 [2], JPMorgan still maintains the Interbank Information Network. The Interbank Information Network is now called "Liink", and involves a partnership of over 400 financial institutions over 78 countries [62].

In 2015, development of the Multichain permissioned blockchain started by Coin Sciences, and is currently in production by multiple enterprises including the pharmaceutical industry [36]. The same year, Coinprism started developing their enterprise OpenChain platform[14]. The Nasdaq and Chain[15] started a pilot in 2015, for transferring shares on the Nasdaq Private Market [63]. Their open-source project has not been updated for several years[16].

[10]https://www.r3.com/
[11]https://www.r3.com/history/
[12]https://www.r3.com/wp-content/uploads/2020/11/Corda_Spunta_Case_Study_R3_Nov2020.pdf
[13]https://consensys.net/quorum/
[14]https://www.openchain.org/
[15]https://chain.com/about/
[16]https://github.com/chain/chain

Another open-source organization with a focus on permissioned blockchains is Hyperledger[17]. Hyperledger is an umbrella project hosted by The Linux Foundation[18] and maintains different types of tools, libraries and DLTs. Some of these may have the status of "seed", meaning the project is still in its infancy, whereas other projects are already production-ready. There are six different DLTs hosted by Hyperledger, all with different implementations for different use cases. Hyperledger Burrow is a permissioned blockchain using the Ethereum Virtual Machine (EVM) with the Tendermint consensus protocol [54], pursuing high throughput. Hyperledger Indy is based on Sovrin [78] and focuses on storing credentials on a DLT in combination with Zero-Knowledge Proofs. Hyperledger Iroha[19] is a DLT aiming for simplicity and distinguishes itself by allowing the configuration of permissions per query and function request. Hyperledger Sawtooth [71] focuses on high modularity by separating the core of the DLT from the applications that run on it. Hyperledger Fabric [3] is a general-purpose permissioned blockchain platform that can be, and already is, used for a wide variety of industry use cases by supporting pluggable consensus and membership services, as well as data privacy. Hyperledger Besu[20] also uses the EVM, is compatible with the Ethereum main network, and is designed with a permission system in mind for consortia. The last two of these DLTs, Fabric and Besu, are analyzed and compared in this thesis.

2.3.1 Hyperledger Fabric

Fabric is completely open source and was designed for permissioned enterprise settings from the start and aims to differentiate itself in a number of different ways. First of all, many components in Hyperledger Fabric are interchangeable for alternative implementations, including the Ordering Service, the Membership Service Provider and the Consensus Protocol. The Fabric framework provides one or more implementations of these components out of the box that can be used. These, as well as other fundamental components, are explained in more detail in Section 3.1. The pluggable consensus protocols that Fabric supports and how they work in detail are discussed in Section 5.1.

Fabric's *chaincode* (i.e. package of smart contracts) is different than most blockchain systems. Each of these chaincodes have their own namespace on the ledger to store their shared state. The underlying DLT of Fabric takes the form of a blockchain in the formal sense as explained in Section 2.1, in which a total order of transactions exists that are grouped into blocks. Before discussing what is unique about Fabric's approach to chaincode, a problem should be introduced which blockchain DLTs aim to solve.

In traditional blockchain DLTs, there exists only a single total ordering of transactions. In case two transactions modify the same value, one should always be executed before the other. This means a new transaction may never be executed, until the previous transaction is finished. This becomes a problem when we consider an invalid transaction that is committed to the ledger, e.g. a transaction that executes functionality that never terminates, such as an infinite loop. Since the halting problem has proven unsolvable by Alan Turing, one does not know when to terminate or wait for the transaction to complete. Using timeouts are unreliable, due to heterogeneity in hardware and workloads impacting the execution time. If no measures are taken to prevent this from happening, such a transaction could break the blockchain as all new transactions are waiting for the faulty transaction to terminate.

Permissionless blockchains (i.e., other implementations than Fabric) solve this problem by using Domain Specific Languages (DSLs) that always terminate by definition and/or by introducing gas fees using a cryptocurrency, as discussed in the following section. Yet, a monetary token may have no other use in a permissioned setting, making them undesirable. Smart contracts in Fabric can be written in general-purpose programming languages, such as Node.js[21], Java[22], or Golang[23], without using gas fees.

[17]https://www.hyperledger.org/about
[18]https://www.linuxfoundation.org/
[19]https://www.hyperledger.org/projects/iroha
[20]https://www.hyperledger.org/projects/besu
[21]https://nodejs.org/
[22]https://www.java.com/
[23]https://golang.org/

Now that we have seen how permissionless frameworks commonly solve this issue, we can look at Fabric's approach approach to preventing invalid transactions from bringing the blockchain to a halt. Fabric's transaction flow does not collect transaction requests in blocks, after which they are executed by all nodes as in traditional blockchains (*order-execute* transaction flow). Rather, transactions in Fabric are executed before they are disseminated into blocks, corresponding to their *execute-order-validate* transaction flow. As a result, a transaction that is put on the ledger will not execute a smart contract again! Instead of requesting the invocation of a smart contract, a transaction in Fabric specifies a list of values to write to the ledger's state[24]. Fabric's transaction flow is discussed in Section 4.1.

Anonymity and data privacy is crucial for any permissioned blockchain. Within Fabric, mechanisms such as channels, private data collections, and/or zero-knowledge proofs[25] are supported by the framework. More details on these concepts are given in Section 6.1.

2.3.2 Hyperledger Besu

The *Ethereum Protocol* [97] enables the execution of smart contracts and trading of its native cryptocurrency called *Ether*. This Ether is used to pay for *Gas* to execute the smart contract and ensure smart contracts do not execute forever. When gas runs out, the transaction's changes are reverted. At the time of writing, Ethereum is the most valued permissionless blockchain that can be used for executing smart contracts in terms of market capitalization (i.e. valuation) of its underlying cryptocurrency[26]. Yet, the Ethereum MainNet still suffers from a poor throughput of about 15 TPS, although this is expected to improve when Sharding is introduced in Ethereum 2.0[27]. In addition, private transactions are not supported by the protocol itself (although it is possible to add application level-encryption on top of the protocol). Thirdly, access and usage of the network cannot be restricted due to the lack of permissioning, as the name permissionless blockchain suggests. Any of these three factors make it infeasible for enterprises to run distributed applications requiring confidentiality on the Ethereum MainNet or on the protocol without any modifications.

Whereas the Ethereum protocol specifies the set of rules of the blockchain platform, *Ethereum clients* are concrete implementations of this protocol and run on actual nodes. To overcome these limitations while still being able to use Ethereum, the Enterprise Ethereum Alliance (EEA)[28] was founded. This organization's goal is to "*drive the use of Enterprise Ethereum and Mainnet Ethereum blockchain technology as an open-standard to empower ALL enterprises*" [29]. To achieve this, they published a specification for Ethereum clients, as an additional wrapper or extension to the Ethereum protocol. They refer to this Ethereum protocol with extended rules as *Enterprise Ethereum*. Enterprise Ethereum enables private transactions and permissioning, while still being able to use the underlying Ethereum protocol. Consequently, an *Enterprise Ethereum Client* implements the Enterprise Ethereum protocol. Because an Enterprise Ethereum client contains (and is therefore compatible with) the regular Ethereum protocol, it can either be used on the MainNet (excluding private data) or for private networks. Figure 2.5 illustrates these concepts.

Hyperledger Besu[30] is an open-source implementation of an Enterprise Ethereum Client written in Java. ConsenSys[31] has been a major contributor to the project and offers additional support through their communication channels. Although Besu is a Hyperledger project and part of the Ethereum Enterprise Alliance by implementing an Enterprise Ethereum Client, ConsenSys refers to Besu as an implementation of the ConsenSys Quorum protocol[32]. Before Besu joined Hyperledger, it was named Pantheon[33].

[24]Instead of `execute createAsset(Asset101, "blue")`, the ledger specifies: `Asset101=blue`
[25]Using Identity Mixer / Idemix: https://hyperledger-fabric.readthedocs.io/en/release-2.2/idemix.html
[26]https://coinmarketcap.com/
[27]https://docs.ethhub.io/ethereum-roadmap/ethereum-2.0/sharding/
[28]https://entethalliance.org
[29]https://entethalliance.org/about/
[30]https://www.hyperledger.org/use/besu
[31]https://consensys.net/
[32]https://consensys.net/quorum/developers/
[33]https://github.com/PegaSysEng/pantheon

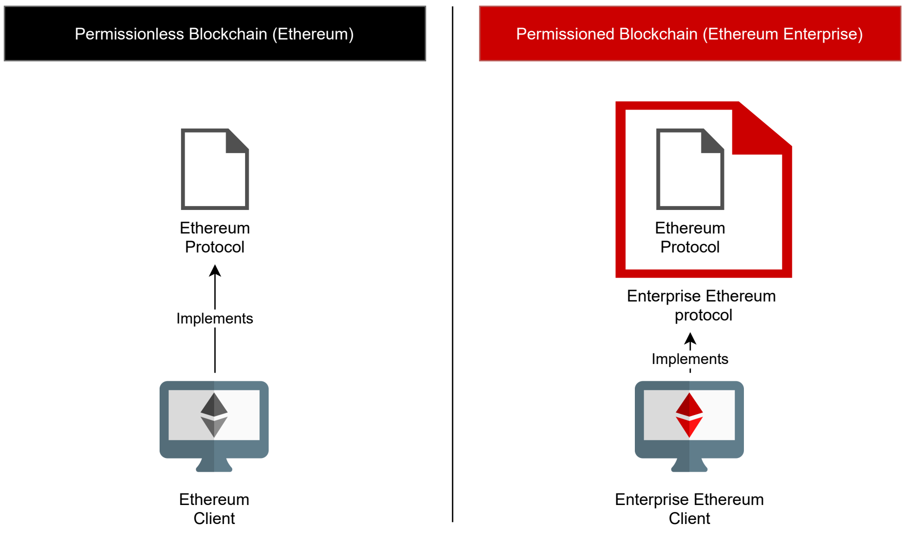

Permissionless Blockchain (Ethereum)
Ethereum Protocol
Implements
Ethereum Client
Permissioned Blockchain (Ethereum Enterprise)
Ethereum Protocol
Enterprise Ethereum protocol
Implements
Enterprise Ethereum Client

Fig. 2.5 Enterprise Ethereum Client

A Besu client may join the MainNet or join a private network. Every network (the MainNet, any of the Ethereum testnets or a private network) has its own blockchain. Like in the Ethereum protocol, public transactions in Besu are recorded on this blockchain and can be accessed by all network participants. For private transactions, however, a smart contract needs to be deployed on a private channel. These private transactions are stored, shared and executed by each node in the private channel using a different application (called Private Transaction Managers) on a different system process. For each private transaction, a cryptographic hash is recorded on the main blockchain to serve as proof the transaction is ordered and has taken place.

This architecture is investigated in Section 3.2. Consequently, how these public transactions are executed is explained in Section 4.2. Besu supports four different consensus algorithms to determine who is allowed to create blocks, including Ethash, Clique, Quorum Istanbul Byzantine Fault Tolerance 1.0, and Istanbul Byzantine Fault Tolerance 2.0. The differences between these consensus protocols are explained in Section 5.2. Lastly, the implementation of data privacy techniques is analyzed in Section 6.2.

2.4 Related Work

Many papers have already been published about Fabric, especially related to studying its architectural implications and suggesting performance improvements. Nasir et al. compared Fabric v0.6 with v1.0 and found a significant increase in performance, but observed it was still slower than traditional database systems [69]. Thakkar et al. proposed optimizations to Fabric v1.0, which were incorporated in Fabric v1.1. Those optimizations increased the transaction throughput from 140 to 2250 TPS [91]. Androulaki et al. provided a high-level architecture of Fabric v1.1 and benchmarked it for a fictional currency [3]. The authors report found a throughput of about 3500 TPS. Gorenflo et al. proposed performance improvements that would increase the transaction throughput of Fabric v1.2 from 3.000 to 20.000 TPS [35], although there has been criticism that some of their optimizations are not practical for production environments. Many of Gorenflo et al.'s changes[34] were suggested for Fabric v2.0, but have not been implemented yet as of Fabric v2.2 [35]. Still, a 2020 paper by Thakkar et al. [90] used Fabric v1.4, used only some of the optimizations suggested by Gorenflo et al., and suggested even more optimizations on top of that. Thakkar et al. claimed to further improve the throughput by a factor of 3. Lastly, Dreyer

[34] https://jira.hyperledger.org/browse/FAB-12221
[35] https://github.com/hyperledger/fabric/releases

et al. compared the performance of Fabric v2.0 with v1 and concluded that the improvements released in v2.0 further improve the throughput and reduce transaction validation times [29].

Androulaki et al. introduced the concept of state-level endorsement policies and privacy-preserving endorsements to hide the identity of transaction verifiers in [4]. Previous work by the author of this thesis investigated the implications for designing endorsement policies for Fabric within the context of supply chains [85, 86].

The literature has already compared several different permissioned blockchains, especially concerning performance and transaction throughput. A 2017 study by Hamida et al. compares Hyperledger Fabric, Multichain, Quorum, OpenChain, Chain Core, Corda and Monax [39]. A 2020 study by Polge et al. compares Hyperledger Fabric, Ethereum, Quorum, MultiChain and R3 Corda [74]. Podgorelec et al. reviewed the different database solutions used by various blockchain platforms, including Fabric and Besu [73]. Although Hasan et al. propose how a logistics system can be implemented in both Besu and Fabric [44], they do not compare both frameworks. However, to the best of our knowledge, no published work at the time of writing this thesis is available aiming to perform an in-depth comparison of Besu and Fabric, nor are there papers that study the architecture and technical details of Besu.

Chapter 3

Architectures

Although blockchain networks are commonly described as a network of peer nodes, often, there are many more components involved, especially as is the case in permissioned blockchain networks. This chapter aims to dissect both Fabric and Besu to their different components. The following research question is answered in this chapter: "***How do the overall architectures of Fabric and Besu differ?***". To that end, Section 3.1 and 3.2 first describe the network architecture of Fabric and Besu, respectively. This section assumes transactions are public on the ledger and disregards any data privacy aspects (until Chapter 6). Lastly, both architectures are compared in Section 3.3.

3.1 Hyperledger Fabric's Architecture

Setting up a network in Fabric involves many moving parts. This section is organized as follows. First, the concept of organizations and trust domain is explained in Section 3.1.1. Using their credentials, organizations can set up an Ordering Service (Section 3.1.2), which initializes the blockchain network. These orderers maintain a Membership Service Provider for managing access to the network, explained next in Section 3.1.3. Each blockchain network always contains precisely one system channel, with additional application channels (Section 3.1.4). Organizations connect to these channels via the Ordering Service using Peers (Section 3.1.5). Each peer maintains its own ledger (Section 3.1.6), which is synchronized using the channels. To modify this ledger, a peer invokes chaincode (Section 3.1.7) to execute transactions. Various ways can be used to invoke this chaincode, including SDKs and gRPC (Section 3.1.8). The endorsement policy (Section 3.1.9) specifies the organizations and roles required to execute these transactions before they can be considered valid.

Two diagrams are shown to guide the reader through the various components. The reader is suggested to consult these throughout the chapter. Figure 3.1 gives a bird's eye view of the components within a Fabric network. The second diagram, Figure 3.2, describes how the various Fabric components are related to each other in an Entity Relationship Diagram. Both images are vector images so that the reader can zoom in. It should be apparent some of the more detailed aspects are omitted (such as policies) to prevent cluttering the diagram. A relationship in the form of "Entity1 (0..1) —defines— (1) Entity2" should be read as: "Entity1 defines exactly 1 of Entity2", or the other way around: "Entity2 is defined by 0-to-1 of Entity1".

3.1.1 Organization and Certificate Authority

A network in Fabric consists of *organizations*. Each organization may represent a single, or an entire group of individuals, like the employees of a company. Organizations themselves are trusted domains, meaning there is complete trust between the members of a single organization. In contrast, however, trust should not be assumed between (members of) different organizations. As a result, an organization's application will prefer requesting ledger information from one of its organization's own members, rather than requesting ledger state from a member of another organization. A *consortium*

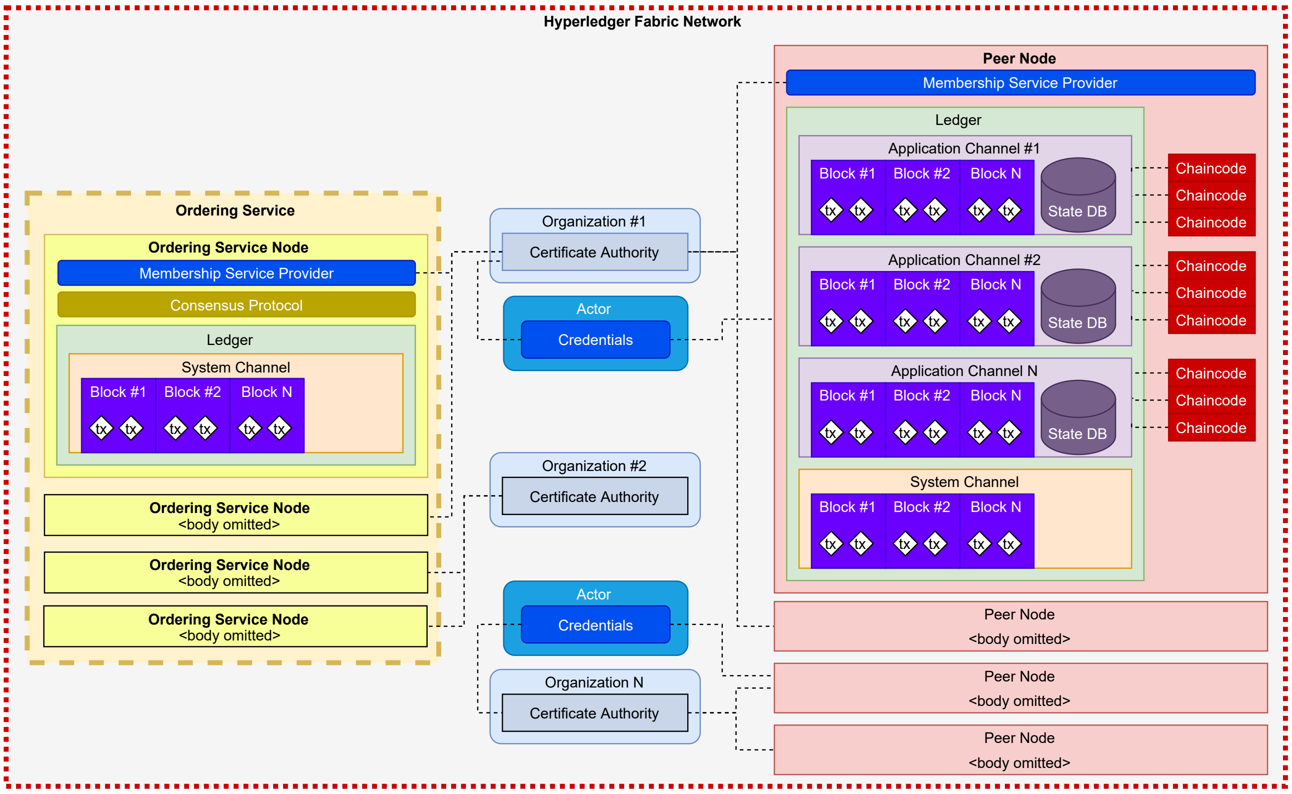


Fig. 3.1 Architecture of a Hyperledger Fabric Network

within Fabric is defined as a group of organizations that wish to collaborate within the network. A network always consists of at least 1 consortium, but may also include multiple (partially overlapping) consortia.

Although members of an organization may be able to identify each other in the physical world, a mechanism is required to identify these members digitally. In Fabric, organizations identify themselves using Certificate Authorities (CAs). If the reader wishes to know more about Public Key Infrastructures (PKIs), message signing, and CAs, the reader is referred to Appendix A.3.

Organizations in Fabric may decide to issue their own root CA, or use an intermediate CA that is signed by another authority (like Comodo or DigiCert). Examples of how organizations may set-up their CA(s) for a Fabric network are shown in Figure 3.3. Thus, it should be apparent that these CAs, certificates and generated digital identities are external to the Fabric network. As a direct consequence, organizations have complete control over managing credentials within the domain of their organization, without having to manage these on (all of their) blockchain networks. Although the certificate domain within an organization is managed outside of Fabric, it does provide a so-called *Fabric CA* for ease of use and testing purposes.

3.1.2 Ordering Service

So far, the described notions of organizations and CAs are external to and do not yet form a network. Rather, the core of a Fabric network is formed by the *ordering service*, which comprises one or multiple *Ordering Service Nodes (OSNs)*. The ordering service has multiple responsibilities related to the configuration, management and operation of a network. Although OSNs may be physically distributed and run by different organizations, they are logically centralized. In essence, this means that the Ordering Service itself can be a decentralized network. In fact, OSNs maintain their own system channel, which is further explained in Section 3.1.4.

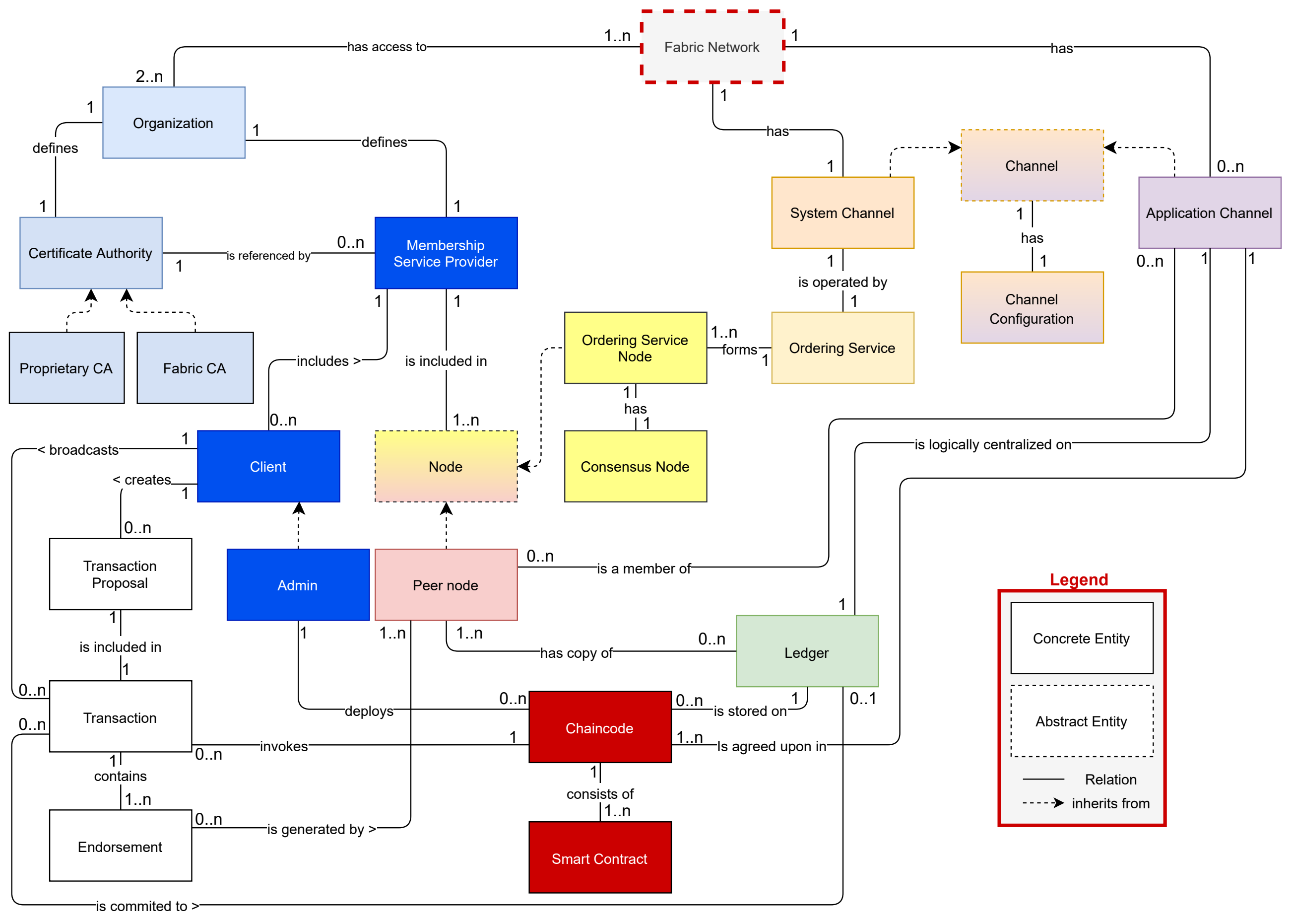


Fig. 3.2 Entity Relation Diagram of Fabric Components and Concepts

The composition of the ordering service is very flexible. Some organizations that are using the blockchain for their business processes may want to have their own OSN. Another organization may only want to have peer nodes without hosting OSNs. At the same time, there may be an organization hosting OSNs, but not part of a business collaboration. Such an organization could be a service provider, regulator, or other legal entity. Note that the ordering service does not *have* to be decentralized and run by multiple organizations. The ordering service could be run by a single organization with only a single node. Although technically possible, this should be avoided for production networks for obvious reasons. To illustrate the flexibility of the ordering service, an example is provided in Figure 3.4.

A primary responsibility of the ordering service is to facilitate the creation of blocks in a deterministic manner. Transactions are not disseminated to all peers within the network, hoping the transaction is included in the next block. Instead, transactions are sent to the ordering service which orders them in blocks and distributes these blocks over the network. By using a consensus protocol to decide on transactions and blocks amongst OSNs before they are shared with the rest of the network, the ledger is always deterministic and final in the sense that no forks are possible. This transaction flow is elaborated on in Section 4.1.

Network administration is another task of the ordering service, especially at the start when no network participants have been defined yet. Although the organization(s) of the OSNs are by default network administrators, other network administrators can be added. Then, any of these network administrators (organization(s) of the ordering service or a different organization that was added as administrator) can define consortia. These are recorded in the network configuration of the system channel, which is maintained by the ordering service. Recall that a consortium is a group of organizations wishing to transact. But before these members of consortia can collaborate, they have to create their own application

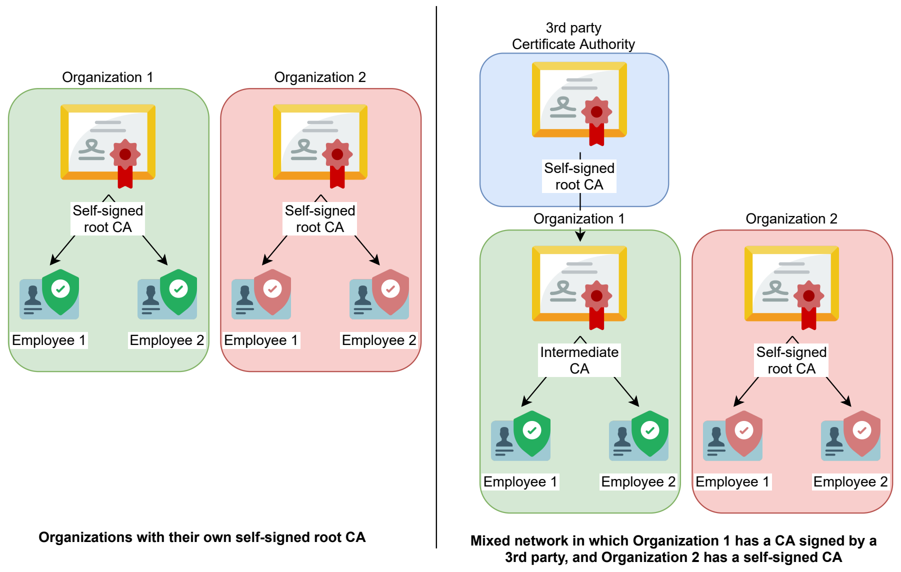


Fig. 3.3 Certification Authorities of Organizations

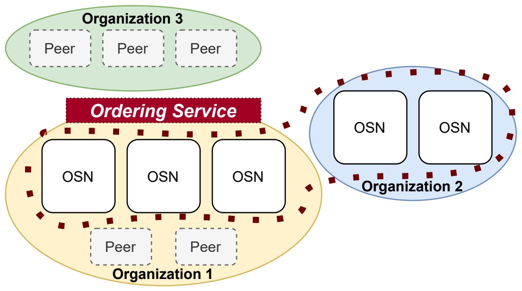


Fig. 3.4 Example of an ordering service

channel themselves and record this to the system channel. Thus, it is apparent that the ordering service is the initial gatekeeper of who is admitted to the network: The ordering service needs to specify additional network administrators or consortia members before the first application channels can be created and collaborations can take place. In Section 3.1.4, Figure 3.6 shows how a network is created and maintained among OSNs using their system channel.

3.1.3 Membership Service Provider

Section 3.1.1 described what organizations are and how these organizations manage their credentials using CAs. Then, we described how an ordering service forms the basis of the network. Recall that an ordering service consists of one or more *organizations*. This is indeed the same notion of an organization as was described earlier, i.e. with their credentials defined by their CA. An OSN takes up the *role* of an *orderer* within an organization. An orderer, however, is simply one of the exclusive roles given by an organization. Other roles include a *peer* for participating in application channels, a *client* for reading the ledger and creating transactions, and an *admin* to perform administrative tasks (such as network administration).

As is in line with the concept of PKI described earlier, all of these credentials are issued by CAs. This means that any organization (even outside of the network!) can create a CA with their own admin roles. However, this does not mean that an organization with a self-proclaimed admin role should have administrative *permissions* in the network! Therefore, a mechanism is required to manage the permissions within the network and on nodes themselves, which is what Membership Service Providers (MSPs) are for. As an example, some organization may have created client and admin credentials by themselves. The client role could be allowed to read the ledger, whereas no admin permissions (of the network) are granted

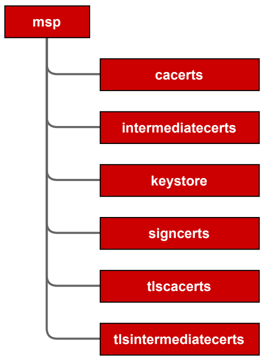


Fig. 3.5 Local MSP file structure

to this organization, despite its credential with an admin role. To summarize, a CA *authenticates* members within an organization, whereas an MSP *authorizes* organizations to participate and perform certain tasks within the network.

There exist two different types of MSPs in Fabric networks: *local* MSPs and *channel* MSPs. The most important difference is the scope and storage medium of the two, described next.

Local Membership Service Provider

As the name suggests, local MSPs are scoped to the local node they are defined on. Every node must have a local MSP. Its implementation is fairly simple, namely a file system directory containing the certificates that are recognized and granted permissions on that node. By looking at the local file structure in Figure 3.5, it should become clearer how local permissioning is realized.

Namely, **cacerts** and **intermediatecerts** contain a list of certificates of the node's own organization, so it knows what nodes to give complete trust. The **keystore** stores a private key that is unique to the node. This node's private key is signed by the node's organization, as testified by a certificate in **signcerts**. Finally, **tlscacerts** and **tlsintermediatecerts** store a list of certificates of *other* nodes in the network, that the node with this MSP trusts. If another organization's Transport Layer Security (TLS)-certificate is not included here, the current node will not trust it and refuses any connection attempts.

Channel Membership Service Provider

A channel MSP defines the organizations that are allowed to connect to a channel and participate in it, as well as those who are allowed administrative permissions for modifying channel configurations. Because it is important to have one logically centralized view among all participants in the same channel, this MSP is defined in the channel configuration itself on the ledger. As a result, there is no confusion between participants who are allowed to participate and who are not. The following section describes in more depth how these channels work.

3.1.4 Channel

All collaborations between participants take place in channels. A *channel* is a system for communication between members of a single consortium while having its own ledger and channel configuration. Channels are analogous to sub-networks. Access to the channel is defined in the channel itself, namely in the channel MSP as explained in Section 3.1.3. By default, *majority voters* [59] (channel administrators) are required to approve channel configuration (block) changes. This shows that all channels support decentralized governance, by requiring the approval of multiple administrative consortia members of a channel.

There exist two main types of channels within a network. Each network has exactly one *system channel* and zero or more *application channels*. All channels (system or application) include a list of orderers for calling the *Deliver* and *Broadcast* function. The former is used to have the OSN deliver a block to a peer, whereas the broadcast function is used to submit an endorsed transaction to the ordering service. Every channel contains a policy specifying what members are allowed to fetch blocks using deliver, and who may submit transactions using broadcast. The policy also specifies who is able to update the channel policy itself.

System Channel

As mentioned earlier, every network contains exactly one system channel. This channel is primarily used for the configuration of orderers, which determines how the network behaves as a system. As opposed to the application channel, the system channel and its ledger is maintained by orderer nodes instead of peer nodes. These orderers are responsible for (1) receiving transactions and processing them into blocks, (2) reach an agreement on the block ordering using a consensus protocol (explained further in Chapter 5.1), and (3) delivering ordered blocks to organizations. The system channel defines what organizations partake in the ordering service.

Like any channel, channel configurations are maintained by the channel administrators. However, since the system channel is used for configuring operations for the entire network, these channel administrators are more commonly named *network administrators*. Although the initial organizations that start the ordering service will by default be (the only) network administrators, these organizations may add other organizations as network administrators afterwards, even if they do not (wish to) host OSNs. An abstract view of this process flow is provided in Figure 3.6

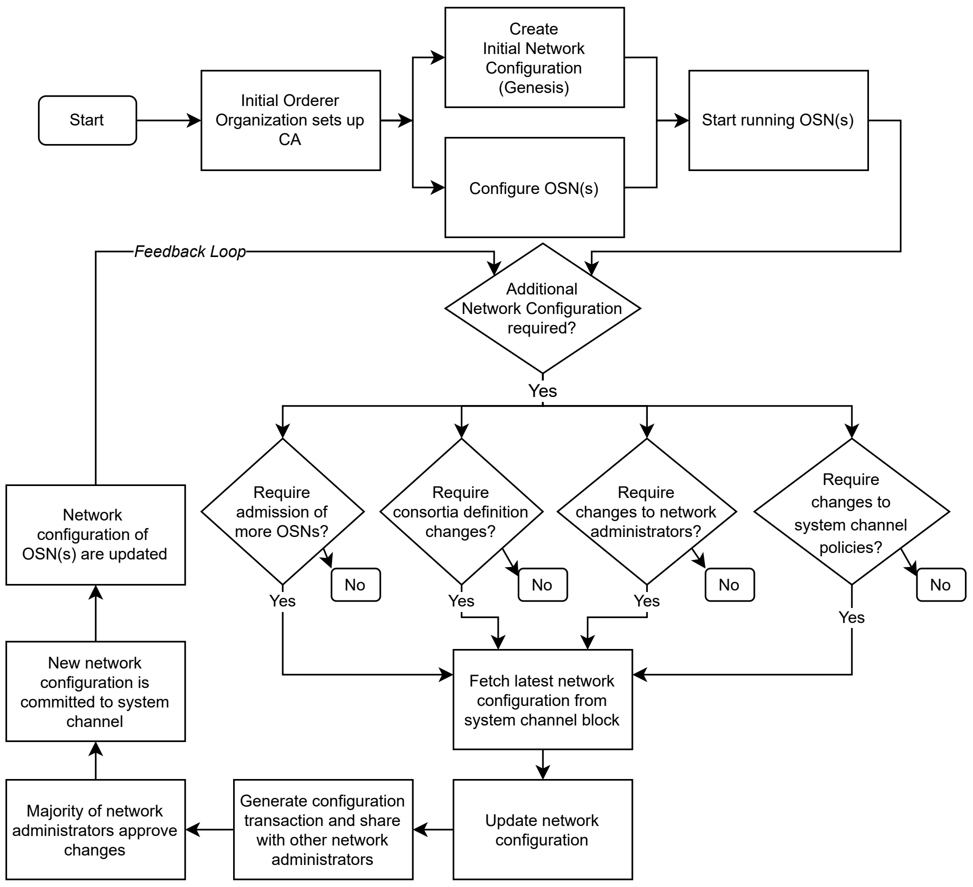

Start
Initial Orderer Organization sets up CA
Create Initial Network Configuration (Genesis)
Configure OSN(s)
Start running OSN(s)
Feedback Loop
Additional Network Configuration required?
Yes
Require admission of more OSNs?
Require consortia definition changes?
Require changes to network administrators?
Require changes to system channel policies?
No
Yes
Fetch latest network configuration from system channel block
Update network configuration
Generate configuration transaction and share with other network administrators
Majority of network administrators approve changes
New network configuration is committed to system channel
Network configuration of OSN(s) are updated

Fig. 3.6 Typical flow of creating and maintaining a Fabric network

Application Channel

As mentioned in the previous section, a channel can only be created by a member defined in a consortia of the system channel. Then, the channel creation transaction is included in the ledger of the system channel, while a new genesis block is created to form the start of the new channel. This genesis block includes the configuration of this new application channel. This configuration includes a consortia definition with a list of organizations that are allowed to participate in the channel. Application channel consortia are completely separated from the consortia definition in the system channel: The *system channel*'s consortia are those who are allowed to *create* a channel, whereas the *application channel* consortia are those who can *participate* in the channel. Consequently, the administrators of a *channel* are in control over what participants may participate, as opposed to network administrators. More importantly, network administrators are unable to join the channel unless explicitly admitted. This makes governance of channels a lot easier when privacy is a concern. By default, the organizations in the system channel consortia are added as administrators of the consortia in a newly created application channel.

Application channels are primarily used by organizations to perform business collaborations using their peers and clients. The rules of these collaborations are defined in chaincode. Channels have various default policies for chaincodes, although these policies may be overridden. Each application channel has its own isolated ledger. Since an organization in one application channel may not be a member of another application channel and not all members can retrieve another channel's ledger state (although some might), transactions across applications channels are not supported. Yet, a peer that is a member of multiple application channels may have both ledgers, and can thus, transact with multiple organizations in different channels (separately). Most of the concepts here (peer, ledger, chaincode, endorsement policies) are only briefly mentioned, as they are explained in later sections.

Relation to Sidechains

Fabric's implementation of channels resemble the notion of sidechains by Back et al., as explained in Section 2.4 [6]. In both architectures, the network is split in sub-networks for groups of business partners to transact privately, i.e. application channels relate to sidechains. In addition, another channel may form the bridge between one or more of these sub-networks, i.e. the system channel or coordination channel for sidechains.

Although Robinson mentioned sidechains may work for many frameworks, the framework needs to satisfy two conditions [80]. The first condition is partially satisfied: it should be possible from one sidechain/channel to read from another sidechain/channel. This is only allowed in Fabric when a peer is a member of both channels. Yet, the second condition is not satisfied by Fabric: a sidechain/channel should be able to make writes to another sidechain/channel. In Fabric, writes to other channels are unsupported as is explained in Section 3.1.7.

Sidechains are actively using coordination channels to transfer data between sidechains, such as a contract of ownership without revealing the purchasing price. In Fabric, the system channel is only used to configure OSNs and to create new channels. The system channel does not have a state database and does not support transactions for business logic (only configuration transactions are allowed).

3.1.5 Peer

While OSNs are crucial nodes for supporting and managing the network, they are external to application channels and their ledgers. Rather, *peer nodes* (peers) form the sub-network of application channels. As such, peer nodes actually store ledgers, create transactions and validate these, whereas the ordering service does not.

Recall that a channel configuration defines *organizations* instead of *peers*, as was explained in Section 3.1.3. Therefore, organizations are flexible in how many peers they want to host and connect to each channel. A collaborating organization may have 0 to many peers in a channel. Having 0 peers may be useful when an organization only has clients that require read-only access through the ledger of other organizations, although there is no guarantee read requests are answered by other organizations, i.e. *multi-tenancy*. This scenario of having 0 peers is supported by the channel configuration as only organizations are defined, rather than individual peers. Still, it is recommended for an organization to have at least 1 peer to eliminate this dependency on others. Another use case for having 0 peers could be an organization that only hosts OSNs.

Moreover, peers can take four different roles: anchor peer, committing peer, leading peer, and endorsing peer. All peers are by definition committing peers, the three other roles are optional. A **Committing Peer** refers to functionality that all peer nodes in the system have in common, of which the validation of transactions in newly received blocks to maintain ledger state is the most important. **Anchor peers** are references to peers that are advertised in the channel configuration, such that network participants of *other* organizations know how to communicate directly to some organization's peer. A **Leading Peer** is used as central point of communication with peers in the *same* organization. In practice, a leading peer fetches blocks from the ordering service and disseminates these blocks under all other peers within the same organization using the gossip protocol. The **Endorsing Peer** is capable of endorsing transactions by *simulating* it, i.e. executing it on a temporary copy of the ledger so the channel-wide ledger's state remains unchanged.

3.1.6 Peer Ledger (LevelDB/CouchDB)

A *ledger* within Fabric is implemented as an immutable, append-only linked-list of blocks. Therefore, Fabric's ledger is an implementation of the traditional *blockchain* DLT (instead of a DAG, hashgraph, et cetera), which was described in Section 2.1. Since all blocks contain an ordered list of transactions and the ledger is a list of ordered blocks, all transactions within the ledger are ordered as well.

The blockchain itself only stores transactions, but not the changes that the transactions describe. Therefore, the ledger also consists of a *world state*. This world state is the most up-to-date representation of the changes resulting from executing transactions. Every application channel has exactly one ledger consisting of a blockchain and a world state. Since the system channel is not used for executing chaincodes, the system channel's ledger does not require a state database. This was confirmed by looking at the file system of an OSN, which verified that a `stateDB` is not present on an orderer.

To highlight the difference between the blockchain (transactions) and the world state (transaction results), consider the following example for the creation and modification of an asset. The blockchain would store 2 transactions in the list, which specify (1) "create asset with key Asset1 and name Foo" and (2) "modify name of asset with key 1 to Bar". The world state, on the other hand, only contains "Asset1 has name Bar", with no knowledge it has ever been "Foo" in the past. Figure 3.7 shows what such a world state may look like. The information on the ledger may be entirely digital such as a document, but may also be just a digital *representation* of a physical asset such as a car. Then, many of the asset's attributes may be omitted by simply uniquely referencing to the physical object. For instance, a car's color, weight, length, brand, and amount of fuel may be replaced by a vehicle's Vehicle Identification Number (VIN).

By looking at the world state, note that the world state is recorded using `(key, value, version)`-tuples. The `key` is used as identifier by chaincodes, whereas the `values` contain additional information on an object. Here, objects are represented using JavaScript Object Notation (JSON) formatting, although this is not required. Any type of String can be stored in the `value` fields. The `version` field is an ever-increasing identifier used for the validation process to prevent the scenario in which two transactions are committed that both modify the same version of the asset (more commonly referred to as double spend). The documentation specifies any versioning scheme may be used, but Fabric uses a tuple of block height and transaction number within block: `(blockNumber, transactionNumber)`.

As we have seen, the world state does not store a history of states, but only the information that results from the last committed transaction. Since all transactions specify the modifications to be performed on the last known ledger state, the most up-to-date ledger state can be (re)computed by starting with an empty ledger, perform validation to ensure the soundness of the next transaction, append it, and repeat until all valid transactions have been added.

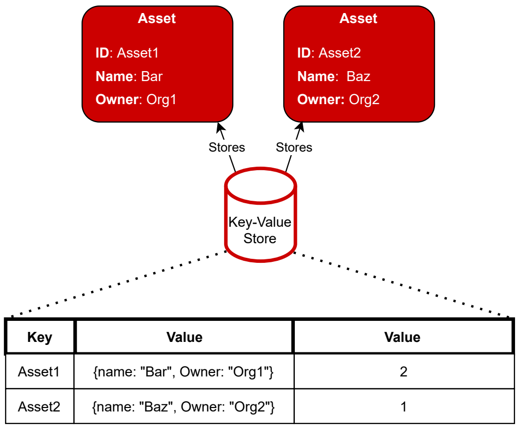


Fig. 3.7 World State

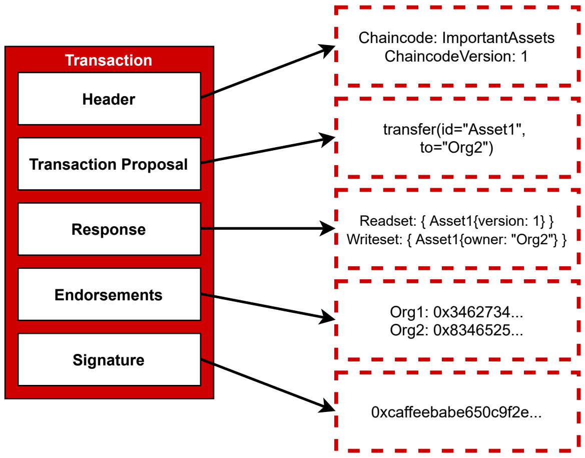


Fig. 3.8 Structure of a transaction with simplified instance

A transaction in Fabric has the structure as illustrated in Figure 3.8. A `header` field is used to specify metadata such as the name and version of the chaincode the transaction executed. The `Transaction Proposal`-field contains the method and arguments of the chaincode invocation. The `Response`-field contains two sets: a readset and a writeset. The *writeset* contains a set of changes that a transaction makes on the key-value store. The *readset* contains a set of versioned keys that the executed transaction depends on, which is used for the validation phase. Essentially, when there is a mismatch between the version of a key in the readset, and the version of the key on the ledger during validation, the transaction is invalidated. Next, the `endorsements` field contains a list of `responses` signed by endorsers that have executed the transaction. These endorsements also contain the original transaction proposal and response, so validators can always verify at a later point in time if these endorsers actually produced the same output as is written in the `response` and `transaction proposal` fields. Finally, the client who submits the transaction to the ordering service signs the transaction and includes a `signature`.

Earlier, we mentioned any type of String can be stored in the `value` fields of the world state. We explained this could be a simple string that represents a single value, or a string representing an object with multiple fields. LevelDB[1] is the default implementation for storing the world state, which stores all data as byte arrays and uses automatic data compression. The only basic operations supported by LevelDB are `Put(key, value`, `Get(key)` and `Delete(key)`. Additionally, LevelDB does not support the client-server model and allows only a single process to access the store simultaneously. From this, it is apparent that LevelDB is very light-weight and low-level implementation of a key-value store. An alternative

[1]https://github.com/google/leveldb

implementation that can be used with Fabric is CouchDB[2]. In contrary to LevelDB, CouchDB can scale from a single node to multi-node clusters and has fault-tolerant mechanisms built-in. CouchDB communicates over HTTP. CouchDB supports JSON-formats natively and allows applications to request and modify these with an arsenal of API-methods. Although CouchDB is much more powerful in terms of features, it comes at no surprise it results in a factor 3-4 lower throughput compared to LevelDB when used in Fabric [50].

3.1.7 Chaincode and Smart Contract

The previous section (3.1.6) already explained how state is stored on the ledger. This state is modified and used by distributed applications. A distributed application on Fabric takes the form of both a *Smart Contract* and a *Chaincode*. Although both terms are used interchangeably by both the documentation and literature on a conceptual level, on a technical level there is a difference. A smart contract is a collection of functions to realize business logic. Chaincode is a package of one or more smart contracts. The difference will become clear after seeing how a chaincode is developed, installed and committed.

The distributed application is written in a general-purpose programming language using smart contract APIs. At the time of writing, the only APIs that are officially supported are Go[3], Java[4] and Node.js[5]. Although there are slight differences between the three APIs after inspecting the chaincode API's source code, *all* of them mainly use three functions: `PutStringState/PutState(Key, Value)`, `GetState(Key)` and `DelState/DeleteState(Key)`. APIs for different programming languages differ slightly in function naming. It is noticeable that the aforementioned operations correspond one-to-one with the available operations in LevelDB as described in Section 3.1.6.

To show the similarities between the various chaincode APIs, smart contract code from the Fabric test network *`asset-transfer-basic`*[6] has been taken, simplified, and shown below to give the reader an impression of how to create an asset in the world state in the various programming languages. These listings are provided in Appendix A.1: Listing A.1 shows asset creation for Java, Listing A.2 for JavaScript (Node.js), and Listing A.3 for Golang. Please note that some important details (such as Asset class definition) have been left out as to focus on the relevant parts. Each method in the smart contracts of all examples start with a `Context` parameter, used to supply the method with a reference to the key-value store to perform the world state operations on.

Packaging Smart Contracts in Chaincode

Once the smart contracts have been written, they need to be packaged in a `.tar.gz` file, which can be digested using a command-line utility provided by Fabric called `peer chaincode lifecycle package`. These packaged smart contracts together form the *chaincode*. While chaincodes may contain multiple smart contracts, they are eventually bundled. According to the developers of Fabric, most use cases only require a single smart contract per chaincode[7].

After close inspection of a packaged chaincode-file, this is an archive containing (1) a `metadata.json` file and (2) a nested `code.tar.gz` source code file. The metadata file seems to have the same three fields regardless of the programming language used: `type` for the programming language of the source code, `label` for a name given to the chaincode, and finally a `path`, a relative path to where the smart contract source was located when it was packaged. The next step is to install the chaincode, i.e. archive, on the peer nodes using a `peer lifecycle chaincode install`-command. So far, everything has been performed off-chain.

[2] https://couchdb.apache.org/
[3] https://github.com/hyperledger/fabric-contract-api-go
[4] https://github.com/hyperledger/fabric-chaincode-java
[5] https://github.com/hyperledger/fabric-chaincode-node
[6] https://github.com/hyperledger/fabric-samples/tree/master/asset-transfer-basic
[7] https://hyperledger-fabric.readthedocs.io/en/release-2.2/developapps/chaincodenamespace.html#considerations

Deployment of Smart Contracts

Once organization administrators have installed the chaincode on their systems, they can send a transaction to the orderer to approve the chaincode on the organization's behalf using the command `peer lifecycle chaincode approveformyorg`. Still, this only says that one organization has approved a chaincode, and not that the chaincode is deployed on the channel. The chaincode may only be deployed network-wide once "enough" organizations have expressed their approval through the aforementioned command. Which or how many organizations must approve the chaincode before being admissible, is specified in the channel configuration (`Application.Policies.LifecycleEndorsement`) and enables a layer of decentralized governance within the channel. By default, this is set to `MAJORITY Endorsement`. This value is also used for upgrading chaincodes. Any organization in the network can check which organizations have approved a chaincode and those that have not (yet) approved it. Once the policy has been satisfied, any organization can now finally commit the chaincode to the channel, making it available for transactions.

When a transaction proposal invokes a smart contract, its locally-installed implementation is executed. This is fairly unique to Fabric, as it is able to execute transactions using different implementations of the chaincode. In fact, organizations are able to execute transactions on slightly different chaincode implementations than originally committed to the channel. To try this out, we conducted an experiment[8] by installing the same chaincode *interface* using two slightly different chaincode *implementations* on two peers by removing some safety checks (if-statements) from the implementation. The default settings of the test network[9] were used. After both peers have approved a slightly different chaincode implementations, majority was reached and the contract could be deployed on the channel successfully. The execution of transactions were also successful (as long as the transaction results are the same), despite being executed by different code. we were also able to deploy the same contract interface with a Java implementation and a Golang implementation, and commit this to the ledger. Executing transactions, however, fails across different programming languages. This might seem as a small experiment just out of curiosity, but it shows a much more important underlying phenomenon: organizations are not bound by the chaincode that was committed, but can adapt or manipulate the chaincode for their own benefit. More on this is explained in Section 4 on transaction flow.

Scope of Chaincodes

Earlier, it was explained in Section 3.1.4 and 3.1.6 how each channel has its own ledger with its own world state. By deploying a chaincode on a channel, a new *namespace* is created within the world state for this chaincode. As a result, the state of chaincodes cannot directly be read or modified by other chaincodes. Since the world state is essentially a key-value store, this isolation can prevent collisions of multiple chaincodes modifying the same values. While two chaincodes in the same channel are unable to directly access each other's state databases, they are able to transact by invoking the other chaincode. This limits a chaincode from only accessing values exposed by the smart contract, as some values of the state database may not be exposed.

Chaincode can only be invoked by members of the channel that the chaincode is installed in. This is explained by the fact that the actual source code is only installed on local machines, and a peer would not respond to a request of an organization outside the channel. While a member of only one channel *cannot* invoke chaincode of another channel, a member of both channels *can* perform read-operations on the other channel using chaincode[10]. For a chaincode to query the chaincode of another channel, it has to invoke a local `invokeChaincode()` function. The reader should be aware that this should be used with caution, because this invocation happens by the general-purpose programming language without being registered in the readset and writeset: Essentially, it would be analogous to fetching a value from any external source off-chain, such as a public Web API or a company's relational database. As such, the returned value may be inconsistent. When two peers validate a transaction by fetching this value from another channel, the other channel's state may have been

[8] https://github.com/Viserius/FabricBesuBenchmark/blob/master/demo_different_CC.sh
[9] https://github.com/hyperledger/fabric-samples/tree/master/test-network
[10] https://hyperledger.github.io/fabric-chaincode-node/release-2.2/api/fabric-shim.ChaincodeStub.html#invokeChaincode__anchor

changed in the meanwhile. Additionally, a cross-channel query is only successful if enough organizations are in the other channel according to the endorsement policy, otherwise not enough peers are able to retrieve this value and endorse the transaction (unless these organizations fetch the required value differently in the chaincode, such as by setting it as a constant). Table 3.1 summarizes these different types of chaincode-to-chaincode communications.

Supported Operations	Chaincode-to-Chaincode Communication	
	Within Same Channel	Across Channels
Has second chaincode installed	Read, Write	Only Read
Does not have second chaincode installed	None	None

Table 3.1 Types of chaincode-to-chaincode communication

3.1.8 Interacting with the network

To interact with a peer, the Fabric team has developed a number of Software Development Kits (SDKs). Currently, SDKs are provided for the Java programming language[11] and for Node.js[12]. In addition, SDKs for Golang[13] and Python[14] are in development, although not officially released yet. After further inspections, these SDKs communicate using the gRPC[15] protocol[16]. The gRPC endpoints are documented in a GitHub repository[17], but not discussed in the Fabric documentation. Using this specification, one could interact with a Fabric peer using a different (unsupported) programming language, although it would take some effort to develop. For manual interaction with the network, a command-line interface is available (also using gRPC). Since this tool provides an "inconsistent experience to the user"[18], there is an ongoing effort to redesign the command-line tool[19] that will use an SDK instead of gRPC. To conclude, all communication with a Fabric peer happens over gRPC. The gRPC endpoints may be communicated to directly, through the command-line interface, or through an SDK.

3.1.9 Policy

So far, a couple of policies have been mentioned, like the channel policies specifying who can deliver blocks and who can broadcast transactions to the orderer. Likewise, we have seen an example of a chaincode lifecycle policy specifying who must approve a change to the chaincode before it may be admitted to the ledger. In fact, these policies follow the same syntax and semantics, but operate on different levels of the network. A policy's grammar, examples, and an interactive demo of Fabric's policies are given in Appendix A.4.

Endorsement Policy

One type of policy particularly important for the validation of transactions is the *Endorsement Policy*. An endorsement policy specifies which organizations must execute a transaction before it can be considered valid. To perform this execution, an organization receives a transaction proposal from a client, invokes the chaincode according to the method and arguments supplied in the proposal, and returns an *endorsement* containing a readset and writeset. For more details on the transaction's structure, the reader is referred to Section 3.1.6. This process of generating endorsements is called *endorsing*.

Endorsement policies apply to the execution of smart contracts, but can be defined on four levels: in the channel configuration, in a chaincode, in a Private Data Collection, or per key-value pair. On the highest level, the channel

[11] https://github.com/hyperledger/fabric-sdk-java
[12] https://github.com/hyperledger/fabric-sdk-node/
[13] https://github.com/hyperledger/fabric-sdk-go
[14] https://github.com/hyperledger/fabric-sdk-py
[15] https://grpc.io/
[16] https://github.com/hyperledger/fabric-sdk-node/blob/master/fabric-common/lib/ServiceEndpoint.js#L180
[17] https://github.com/hyperledger/fabric-protos
[18] https://jira.hyperledger.org/browse/FAB-10734
[19] https://github.com/hyperledger/fabric-cli

configuration contains a default endorsement policy. When a chaincode is deployed without specifying an explicit endorsement policy, the default of the channel will be used. Zooming in, an endorsement policy can be specified on the chaincode-level, allowing different chaincodes to have different endorsement policies, e.g. when the chaincode is only used by a subset of organizations in the channel. Still, this chaincode-level endorsement policy is not very flexible, as updating the policy requires a new version of the chaincode to be deployed. Although Data Privacy is introduced in Chapter 6.1, we briefly mention that an endorsement policy can be set per private data collection to override the chaincode-level endorsement policy.

Common examples in the literature are settings in which multiple organizations trade assets, where *only* the organizations involved in the transaction are required to endorse the transaction [4, 86]. This cannot be achieved efficiently using chaincode-level endorsement policies (explicit or implicit). For instance, `OR(AND('Org1.member', 'Org2.member'), AND('Org2.member', 'Org3.member')` would still allow `Org1.member` and `Org2.member` to endorse a transaction that involves `Org3.member` without its endorsement. Since Fabric 1.3, state-based endorsements were introduced. State-based endorsements can be modified through regular chaincode invocation, as opposed to having to update the entire chaincode. State-based endorsements are stored using *state-level metadata*. Essentially, every single key in the chaincode's namespace of the world state may have its own endorsement policy specified.

This section mentioned four different ways of specifying an endorsement policy: in the channel configuration, on the chaincode level, in private data collections, or on the state-level. In case multiple endorsement policy specifications are applicable, the most specific endorsement policy is applicable (as stated here from more general to more specific). In general, coming up with a single endorsement policy on any level is nontrivial: An endorsement policy that is too permissive by requiring few endorsements is subject to colluders manipulating the ledger state, whereas an endorsement policy that requires too many endorsements requires the involvement of too many organizations hurting data privacy [86]. Thus, there is often a trade-off between ledger integrity and data confidentiality.

3.2 Hyperledger Besu's Architecture

The background information in Section 2.3.2 explained how Besu is an *Ethereum Client*, meaning it is one of many implementations of the *Ethereum Protocol* (hereafter simply referred to as Ethereum). On top of this, Besu extends Ethereum by also implementing the *Enterprise Ethereum Alliance Protocol* to provide additional functionality including private transactions and permissions. Since the majority of Besu's architectural components and concepts follow Ethereum, this also forms the basis for this section. Still, Ethereum is only a protocol and does not have a single, concrete implementation (Go Ethereum[20] is often the go-to standard, but other implementations like Open Ethereum[21] also exist). It is important to point out that Besu does not fork any of the other Ethereum implementations, but provides its own implementation written in Java. Therefore, the remainder of this work focuses on the implementation provided by Besu, which may or may not be different from other Ethereum implementations.

Similarly to our approach for Fabric, a diagram presenting the overall architecture of a Besu network is shown in Figure 3.9. Likewise, common components and concepts that appear throughout the chapter are shown in the Entity Relationship Diagram in Figure 3.10, highlighting how the various components are related to each other. Both figures may help the reader grasp the bigger picture, while going through the chapter.

3.2.1 Node

A Besu network is run completely by homogeneous peer nodes that run an instance of the Besu Enterprise Ethereum Client. Communication between peers are completely peer-to-peer, without involving different node types or requiring partial centralization. These nodes are called *Full Nodes*, which store the entire blockchain, including the complete ledger with all

[20] https://github.com/ethereum/go-ethereum
[21] https://github.com/openethereum/openethereum

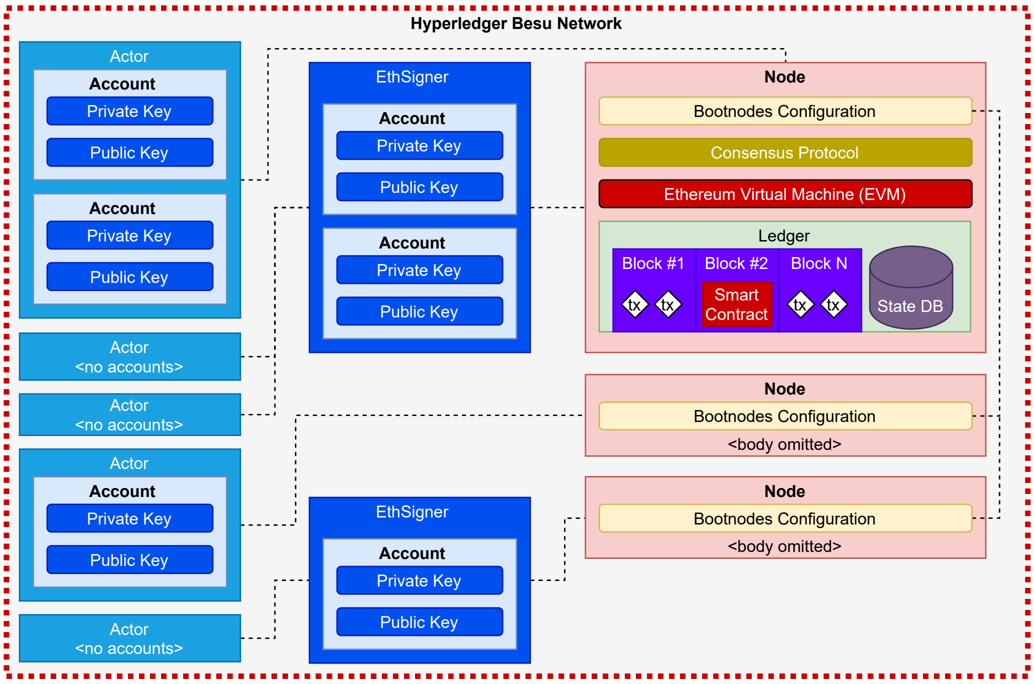


Fig. 3.9 Hyperledger Besu Network Architecture

historic transactions. Since full nodes have to execute all transactions sequentially (i.e. fast synchronization is unsupported), they also have the latest world state. New full nodes that join the network are able to derive this world state as well by requesting blocks from other peers. However, since full nodes only maintain the most up-to-date world state, full nodes cannot quickly retrieve exact states at any previous block in the past (such as smart contract state and balances at a block prior to the current head).

Although it was previously mentioned the network is run by homogeneous nodes, there is a little more nuance to it. *Archive Nodes* exist as an extension of full nodes, by additionally storing every single intermediate world state for each block, which allows any historical world state to be queried without recomputing it from transactions. Consequently, archive nodes take up a lot more disk space[22]. All full node functionality is also available to archive nodes.

Another topic of interest is the relationship between organizations and nodes. This relationship in Besu is very simplistic, as the concept of organizations does not exist. All nodes have their own public and private key, which is how they authenticate themselves. This also means organizations are flexible in how they wish to participate in the network: a (small) organization may wish to participate using only 1 node, whereas another (larger) organization may wish to participate using multiple nodes, e.g. one per department.

Network Creation

Creating and starting a network is fairly simple in Besu and involves three main steps. First, a genesis-file needs to be configured which is used to create the genesis block. Then, node-specific configuration has to be set in a `config.toml`-file. Lastly, the nodes (including one boot node) should be started. This process of setting up a network is explained below in more detail, and is also illustrated in Figure 3.11.

The genesis configuration file is relatively small and contains various settings that apply to the whole network. Four configuration values are of particular interest to permissioned settings: `config`, `gasLimit`, `contractSizeLimit` and `alloc`. The first, `config`, specifies the consensus protocol used in the network. `GasLimit` specifies the maximum amount

[22]Just to give an example, for the Ethereum Mainnet, a full node may store about 400-700 GB of ledger data, whereas an archive node stores about 6.5 TB of ledger data

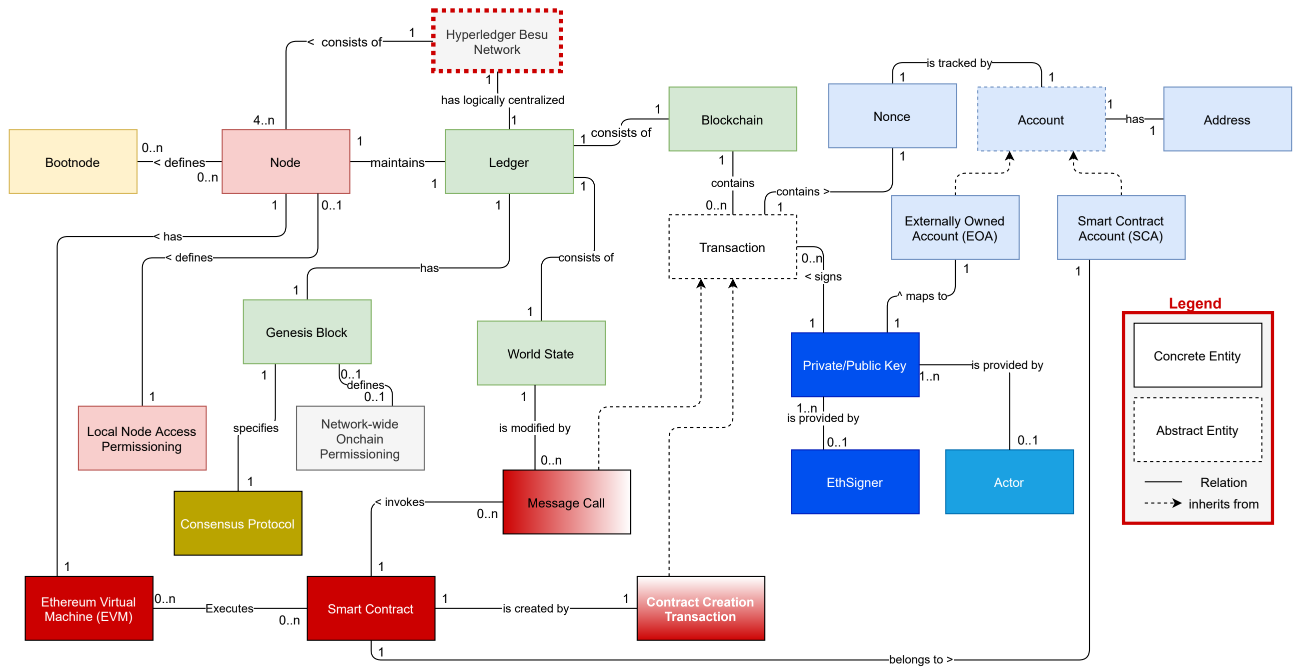


Fig. 3.10 Entity Relation Diagram of Besu Components and Concepts

of gas of a block. In an enterprise setting, this may be set to the maximum value (`0x1fffffffffffff`) to allow for significantly more transactions in each block[23]. The `contractSizeLimit` puts a hard limit on the size of a contract that may be deployed at a default of 24 kilobytes and may be increased to a maximum to allow for larger enterprise applications[24]. Finally, `alloc` may be used to predefine smart contracts or externally-owned accounts with an Eth balance in the genesis file.

Next, the individual nodes should be configured. By briefly going through this configuration, the reader is given an idea of how small the configuration is. Almost all configurations apply to features that can be toggled on or off. `rpc-http-enabled`, `rpc-ws-enabled`, and `graphql-http-enabled` together with host and IP options specify if this node exposes its functionality over JSON-RPC, WebSockets, or GraphQL. `Metrics-enabled` allows applications such as Prometheus[25] to (regularly) poll blockchain metrics such as its height, last block number, connected peers and the maximum amount of peers allowed to connect to that particular node. Each node can also restrict access by accounts and nodes locally through a `permissions-nodes-config-file`, which is further explained in Section 3.2.7. Especially important for a permissioned network is that nodes explicitly set their `min-gas-price` to `0`, otherwise nodes only execute transactions with a sufficient amount of gas. Lastly, each node specifies a list of `bootnodes`. Although the name might suggest differently, *bootnodes* are not components or nodes themselves, but merely references to other nodes for *peer discovery*. While it may seem the network would become centralized around these bootnodes as gatekeepers, this is not the case as bootnodes are like any other node with the exact same permissions. Consequently, peers may swap their locally configured bootnode for any other node in the network.

Once a minimum number of peers (depending on the consensus protocol) have found each other through bootnodes, the network is up and running. An example of how various organizations can collaborate within a network in terms of peers is shown in Figure 3.12. Organizations are colored for the reader's clarity and do not represent a technical difference, i.e. recall that Besu technically does not know the concept of organizations. Yet, organizations may choose one of their own nodes as main bootnode which is connected to the rest of the network (Org3 in the diagram). From this diagram, it should

[23] https://besu.hyperledger.org/en/stable/HowTo/Configure/FreeGas/#1-set-the-block-size
[24] https://besu.hyperledger.org/en/stable/HowTo/Configure/FreeGas/#2-set-the-contract-size
[25] https://prometheus.io/

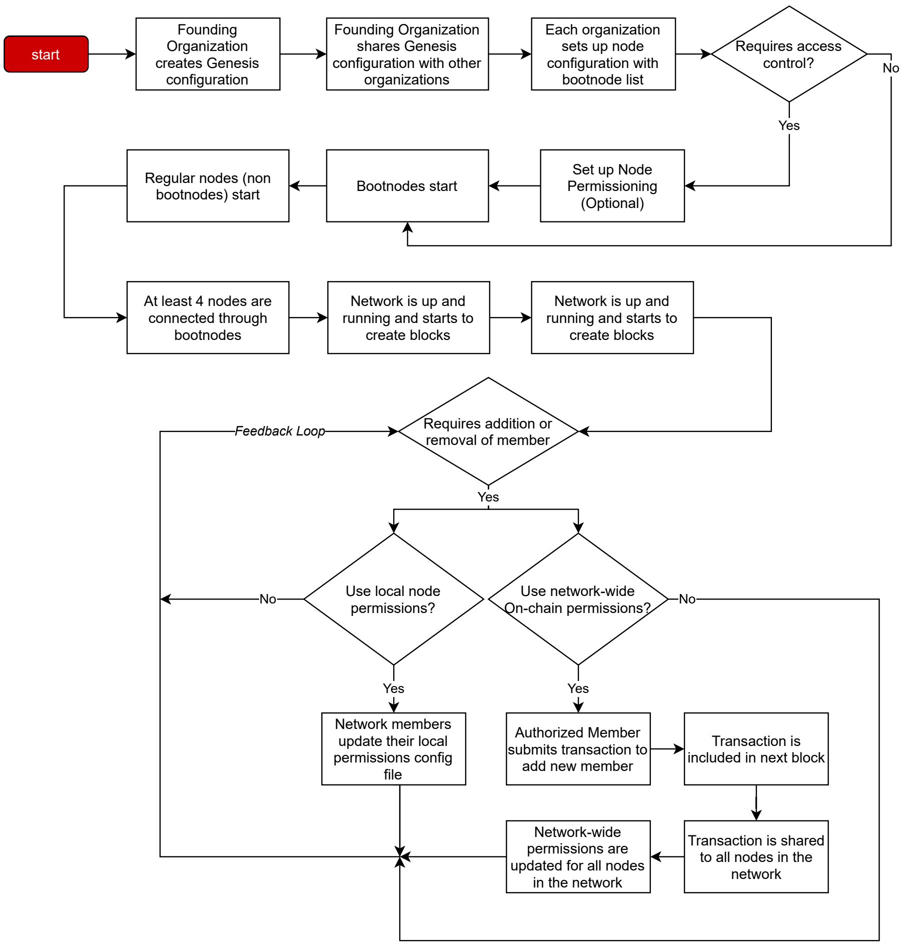

start
Founding Organization creates Genesis configuration
Founding Organization shares Genesis configuration with other organizations
Each organization sets up node configuration with bootnode list
Requires access control?
No
Yes
Set up Node Permissioning (Optional)
Bootnodes start
Regular nodes (non bootnodes) start
At least 4 nodes are connected through bootnodes
Network is up and running and starts to create blocks
Network is up and running and starts to create blocks
Requires addition or removal of member
Feedback Loop
Yes
Use local node permissions?
No
Use network-wide On-chain permissions?
No
Yes
Yes
Network members update their local permissions config file
Authorized Member submits transaction to add new member
Transaction is included in next block
Transaction is shared to all nodes in the network
Network-wide permissions are updated for all nodes in the network

Fig. 3.11 Typical flow of creating and maintaining a Besu network

also become clear that organizations are flexible on specifying their bootnodes and may even specify multiple bootnodes across multiple organizations (Peer of Org2 points to one of their own organization's peers and a peer of Org3). As long as it points to at least 1 node connected to the network, it can communicate to all other nodes through *peer discovery*.

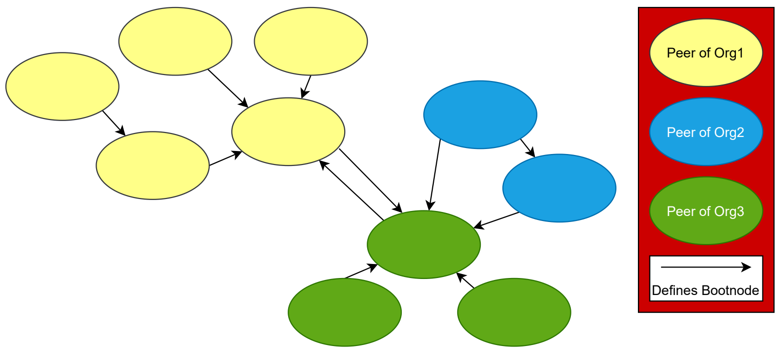


Fig. 3.12 Example network composition of three organizations

3.2.2 Account

Section 3.2.1 described what nodes are, how they connect to each other and how they form the network. Yet, these nodes do not participate in the creation or signing of transactions. Their only role is to keep track of the ledger, create blocks and validate transactions. Transactions are created and signed using *Accounts*. In fact, these are entities that may be completely unaffiliated to the nodes in the network. Transactions are created *by* accounts and *between* accounts. This is further explained in Section 3.2.3. In summary, it is important to understand nodes are nothing more than the *infrastructure* of the network, whereas accounts are the *entities* acting within the network. There exist two types of accounts: externally owned accounts and smart contract accounts. Both types of accounts are able to interact with smart contracts.

Externally Owned Account

An *Externally Owned Account (EOA)* is an account that is typically owned by an individual or group of individuals. Essentially, an EOA is a pair of a public and private key, and is completely external to the network itself. Therefore, an account can be created by anyone and may be used on any Ethereum-based network provided the account has been explicitly granted access (or no permissions are in place for the network). The public-private keypair is generated using elliptic-curve cryptography, which is a faster and more secure approach to cryptography than Rivest-Shamir-Adleman (RSA) [41].

To create an account, one would first randomly generate a private key by taking a random integer between `0` to about 2^{256}. This range corresponds to about 10^{77} possible numbers, which is only a factor of 100 to 1000 less than the amount of atoms in the observable universe[26], highlighting how low the chances are of choosing the same private key (i.e., *collision*) provided a sufficiently good pseudo-random generator is used. The private key is used to derive a public key. The `secp256k1`-algorithm [18] is used to compress this public key to a single integer, often represented as a hexadecimal string representing the address of the account.Using these public and private keys, transactions can be signed using the Elliptic Curve Digital Signature Algorithm (ECDSA) (instead of using a PKI). Consequently, participants are not required to share public keys with each other. This is exactly the reason why EOA are created off-chain, without requiring the public key to be registered on the ledger beforehand.

Smart Contract Account

A Smart Contract Account (SCA) is automatically created when a smart contract is being deployed by an EOA. A SCA also has its own address, which is deterministically generated by encoding the EOA's address and nonce, after which they are passed to `keccak256` hash function[27]. This address of a SCA is important for the network, as it is the account's main identifier and allows other accounts to find and interact with it. More information regarding smart contracts is given in Section 3.2.4.

Keypair Structure

Although it seems that every single node and account in Besu has exactly 1 public/private keypair, we investigated the exact structure of this keypair to see if they could be signed by a common parent entity, e.g. organization. To investigate this, we dived into the cryptographic implementation of Besu to see how exactly these keypairs are used and stored. As we saw earlier in Section 3.2.2, Besu uses ECDSA for generating the public/private keypair. It turns out, however, that the public and private keys are always used as plain text: the private key is a 64-digit hexadecimal integer, corresponding to a 256-bit integer. The public key is generated using the `secp256k1` elliptic curve and is a 128-digit hexadecimal integer, corresponding to a 512-bit integer. More importantly, these keys are always used numerically, rather than a certificate (such as X.509) as is used in Fabric. Because keys in Besu are not wrapped by a certificate, they cannot be signed by an overarching organization.

[26]https://en.wikipedia.org/wiki/Eddington_number
[27]https://github.com/ethereum/eth-hash

3.2.3 Ledger

In order to understand how data is being stored in Besu, we look at Besu's source code since the documentation is very limited. We find that `StorageProvider`[28] is used as facade for creating the ledger and is shown in Listing 3.1.

```java
public interface StorageProvider extends Closeable {
  BlockchainStorage createBlockchainStorage(ProtocolSchedule protocolSchedule);
  WorldStateStorage createWorldStateStorage(DataStorageFormat dataStorageFormat);
  WorldStateStorage createPrivateWorldStateStorage();
  GoQuorumPrivateStorage createGoQuorumPrivateStorage();
}
```

Listing 3.1 StorageProvider

First of all, the `BlockchainStorage`[29] is used for storing blocks. This facility exposes `put`-operations for storing blocks by providing the hash and body of a block. These blocks are stored using a pluggable key-value store implementation. By default, this is set to *rocksdb*[30], although one can change this by developing a plugin for another key-value storage implementation. The `BlockchainStorage` also stores a mapping from block number to block hash, such that one can quickly retrieve the *n*-th block in the chain.

Secondly, we see that the world state is stored using a `WorldStateStorage`[31] facility. The world state is stored using a *trie*-data structure, also persisted to the set key-value storage implementation. By default, however, a slightly adapted *Bonsai Trie* [26] data structure is used. The option to modify this value also seems to be hidden and can thus be considered the recommended data structure. Perhaps more importantly is what type of data is being stored in this (Bonsai) trie. As can be seen in Listing 3.2, the world state includes a mapping from the accounts address to the account's state, described later. It also contains a mapping of (smart contract) hashes, i.e. addresses, to code. In fact, this code is the smart contract itself. Lastly, the world state also contains a mapping from account hash to account storage. Storage refers to a smart contract's own and unique key-value mapping.

```java
public interface WorldStateStorage {
  interface Updater {
    Updater putAccountStateTrieNode(Bytes location, Bytes32 nodeHash, Bytes node);
    Updater putCode(Hash accountHash, Bytes32 nodeHash, Bytes code);
    Updater putAccountStorageTrieNode(Hash accountHash, Bytes location, Bytes32 nodeHash,
        Bytes node);
  }
}
```

Listing 3.2 Storing of World State in Besu

Thirdly, we see that the function `createPrivateWorldStateStorage()` returns the same data type (`WorldStateStorage`). From this, we know that private data has its own world state and is completely separated from the network's public world state. The last function, `createGoQuorumPrivateStorage()` may be used to ensure that the world state of Besu is stored in the same format as that of another Enterprise Ethereum client called GoQuorum[32] and is out-of-scope for the remainder of this paper. Figure 3.13 shows a simplified view of Besu's ledger.

[28] https://github.com/hyperledger/besu/blob/master/ethereum/core/src/main/java/org/hyperledger/besu/ethereum/storage/StorageProvider.java

[29] https://github.com/hyperledger/besu/blob/master/ethereum/core/src/main/java/org/hyperledger/besu/ethereum/chain/BlockchainStorage.java

[30] https://rocksdb.org/

[31] https://github.com/hyperledger/besu/blob/master/ethereum/core/src/main/java/org/hyperledger/besu/ethereum/worldstate/WorldStateStorage.java

[32] https://consensys.net/quorum/

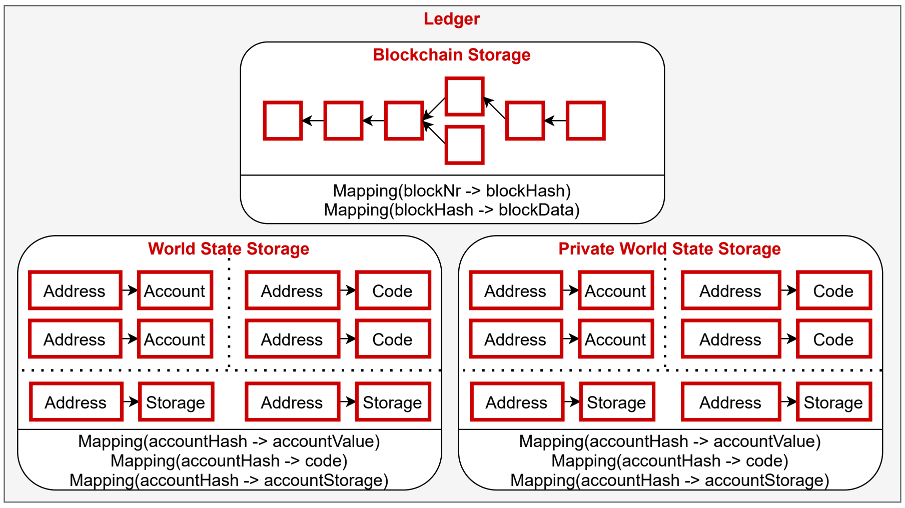

Ledger
Blockchain Storage
Mapping(blockNr -> blockHash)
Mapping(blockHash -> blockData)
World State Storage
Address
Account
Code
Storage
Mapping(accountHash -> accountValue)
Mapping(accountHash -> code)
Mapping(accountHash -> accountStorage)
Private World State Storage

Fig. 3.13 Visualization of data stored in the ledger

Account Structure

Based on the Ethereum whitepaper [97], we already gave an overview of the various account types in Section 3.2.2. However, this does not imply accounts in Besu are conceptually equivalent to the Ethereum protocol. Therefore, we look at how accounts are stored in Besu in terms of their structure on the ledger, especially the `accountState` we saw in the previous paragraphs. Listing 3.3 shows the simplified interface of `AccountState`.

```java
public interface AccountState {
    Hash getAddressHash();
    long getNonce();
    Wei getBalance();
    default boolean hasCode();
    Bytes getCode();
    Hash getCodeHash();
    UInt256 getStorageValue(UInt256 key);
}
```

Listing 3.3 Account State

From this code snippet, we see all accounts have fields for an *AddressHash*, which is the address used to perform transactions. Then, every account is associated with a `Nonce`. As we will see later, all transactions are signed/initiated by a single EOA. Every EOA's transaction includes a nonce that must be greater than the account's nonce in order to be considered valid and prevent double spends. A SCA's nonce is only incremented when a contract deploys a new (other) contract. Next, we see all accounts have a `Balance` in `Wei`, which corresponds to the cryptocurrency and is often not required for permissioned networks. It also becomes clear that accounts include functionality for accessing the account's `code` and `storage`, although these are stored separately and accessed through the account's hash.

The interface of the account above is applied to both EOA and SCA. Yet, an EOA does neither store any code, nor does it have its own (key-value) storage. Code and storage are only present for smart contracts. Thus, we can visualize the structural difference between an EOA and a SCA account in Figure 3.14. Fields in grey are empty and not used, or not relevant for a permissioned setting.

Transaction Structure

Now that it is clear how the ledger is structured, what accounts are and how they are structured, we can look at how accounts suggest modifications to the ledger through *transactions*. A transaction implements the functions shown in Listing 3.4.

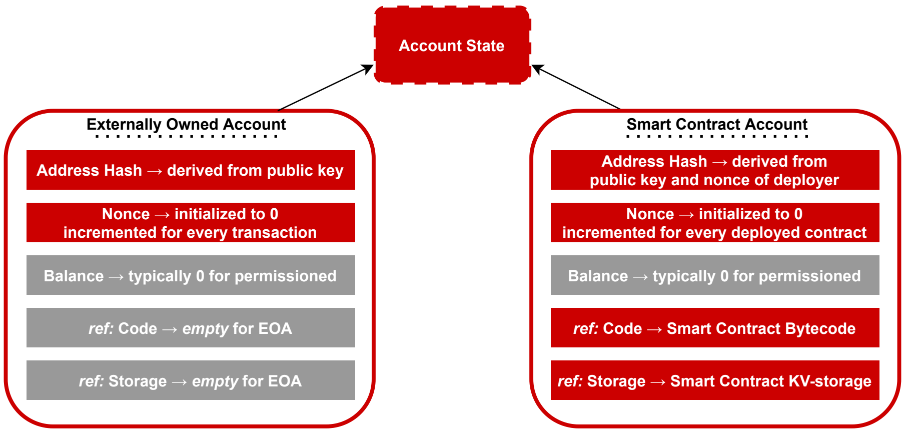

Account State
Externally Owned Account
Address Hash → derived from public key
Nonce → initialized to 0 incremented for every transaction
Balance → typically 0 for permissioned
ref: Code → empty for EOA
ref: Storage → empty for EOA
Smart Contract Account
Address Hash → derived from public key and nonce of deployer
Nonce → initialized to 0 incremented for every deployed contract
Balance → typically 0 for permissioned
ref: Code → Smart Contract Bytecode
ref: Storage → Smart Contract KV-storage

Fig. 3.14 Structual difference between EOA and SCA

This interface documents that Besu distinguishes from two different transaction types, namely *message calls* and *contract creation*.

```java
public interface Transaction {
    Hash getHash();
    long getNonce();
    Quantity getGasPrice();
    Optional<Quantity> getGasPremium();
    Optional<Quantity> getFeeCap();
    long getGasLimit();
    Optional<? extends Address> getTo();
    Quantity getValue();
    BigInteger getV();
    BigInteger getR();
    BigInteger getS();
    Address getSender();
    Optional<BigInteger> getChainId();
    Optional<Bytes> getInit();
    Optional<Bytes> getData();
    Bytes getPayload();
    TransactionType getType();
}
```

Listing 3.4 Transaction Structure

All transactions have their own `hash` for identification and a `nonce`. The transaction's nonce is a (long) number referring to the transaction *creator*'s nonce, i.e. the amount of transactions an account has created thus far. If this transaction's nonce is smaller than the account's nonce, a transaction is invalidated because it would have to be executed before a transaction that has already been included in a prior block or another transaction earlier in the same block. Since all transactions are sequentially ordered and final, a later transaction cannot be executed before an already-included transaction without forking the blockchain. Gas is typically free in a permissioned setting, so discussions about `GasPremium`, `FeeCap`, `GasLimit`, `Value` are omitted. Moreover, every transaction includes `V`, `R`, `S`-values generated by the ECDSA as part of the signing process. The `Sender`-field is the address of the account creating the transaction. Since these values are derived from the `V`, `R`, `S`-values, it becomes clear that the sender refers to an EOA and not a SCA. The `ChainId` is also derived from the `V`-value to prevent reentrancy attacks.

To invoke a smart contract, the sender submits the address of the SCA in the `To`-field with the function call and arguments in the `Payload`-field. A contract creation transaction does not specify a `to`-address, since this address is derived from the sender's address and nonce. A contract creation transaction specifies the bytecode of the smart contract in the `Payload`-field, as opposed to invocation arguments. The `Init`-field refers to the payload field if the transaction creates a smart contract, and is empty for a message call. Likewise, the `Data`-field refers to the payload field if the transaction is a message call, and is empty for contract creation. Finally, the `Type` refers to an `enumerated` type, currently set to `EIP1559`[33] and is used for forward compatibility of new types of transactions[34]. An overview of the two types of transactions is given in Figure 3.15.

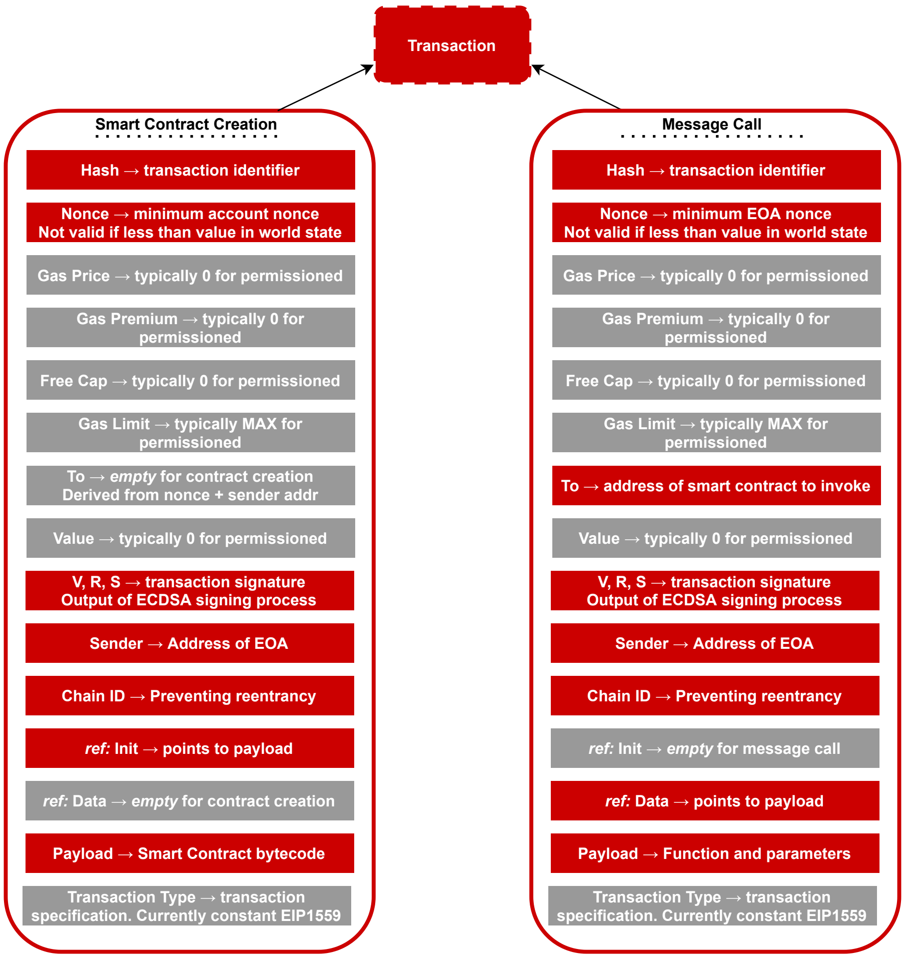

Transaction
Smart Contract Creation
Hash → transaction identifier
Nonce → minimum account nonce Not valid if less than value in world state
Gas Price → typically 0 for permissioned
Gas Premium → typically 0 for permissioned
Free Cap → typically 0 for permissioned
Gas Limit → typically MAX for permissioned
To → *empty* for contract creation Derived from nonce + sender addr
Value → typically 0 for permissioned
V, R, S → transaction signature Output of ECDSA signing process
Sender → Address of EOA
Chain ID → Preventing reentrancy
ref: Init → points to payload
ref: Data → *empty* for contract creation
Payload → Smart Contract bytecode
Transaction Type → transaction specification. Currently constant EIP1559
Message Call
Hash → transaction identifier
Nonce → minimum EOA nonce Not valid if less than value in world state
Gas Price → typically 0 for permissioned
Gas Premium → typically 0 for permissioned
Free Cap → typically 0 for permissioned
Gas Limit → typically MAX for permissioned
To → address of smart contract to invoke
Value → typically 0 for permissioned
V, R, S → transaction signature Output of ECDSA signing process
Sender → Address of EOA
Chain ID → Preventing reentrancy
ref: Init → *empty* for message call
ref: Data → points to payload
Payload → Function and parameters
Transaction Type → transaction specification. Currently constant EIP1559

Fig. 3.15 Structual difference between contract creation and message call transactions

3.2.4 Smart Contract

As we saw in Section 3.2.3, we saw that smart contracts (or SCA) consist of three main components: account state with an address and nonce, smart contract bytecode, and smart contract storage. That section also described how these three main components are stored and connected with each other on the ledger. Although each smart contract is isolated in terms of storage (or state), smart contracts are *composable*, in the sense that one contract is able to invoke another contract. Note however, that even composed smart contracts have to be invoked by an EOA and cannot invoke themselves based on an event. Then, after a smart contract has been invoked by an EOA, this invoked contract can invoke others. The remainder of this section focuses more how smart contracts are created and maintained on a conceptual level.

[33]https://github.com/ethereum/EIPs/blob/master/EIPS/eip-1559.md
[34]https://github.com/ethereum/EIPs/blob/master/EIPS/eip-2718.md

Writing Smart Contracts

As we saw earlier, smart contracts are created using bytecode. The term *bytecode* is commonly used to refer to the *contract creation bytecode*, which includes the constructor of the smart contract[97]. Then, this bytecode is translated to a *runtime bytecode* that is actually stored on the ledger and is used for executing transactions. To deploy a smart contract to the ledger, a contract creation transaction, as we have seen in Section 3.15, needs to be submitted that contains the *contract creation bytecode*. A developer usually does not require knowledge of the *runtime bytecode*, although its existence is good to mention to avoid any possible confusion.

Fortunately, bytecode is not written by a developer by hand most of the times, but generated by compiling higher-level languages. Currently, Solidity, Vyper and Yul(+) are supported by the EVM[35]. All of these languages are DSLs. By far, the most popular language is *Solidity*[36] which is statically typed and heavily influenced by JavaScript. Solidity smart contracts can use libraries defined in supplemental files for reusability and organization of larger codebases. Vyper[37] is another DSL with simplicity, auditability and security as its main goals. Thus, Vyper could be a preferred language when a codebase needs to be critically reviewed by others, which could be the case for mutually distrusting organizations. This, however, does not mean that Solidity cannot be audited. It simply means that smart contracts written in Solidity could be more ambiguous, whereas Vyper's code is more restrictive. A third DSL used for writing smart contracts is Yul[38] or Yul+[39], the latter being a slightly extended version of the former. These are very low-level languages for those wishing to implement their own optimisations. For the remainder of this paper, Solidity is used.

After a smart contract has been written in any of the aforementioned languages, a compiler for that language is used to translate the contract to bytecode. This bytecode is then included in a contract creation transaction, which can be submitted by any account (with access permissions if they are in place) to the network. More information on how transactions are submitted is provided in Section 3.2.6.

Maintaining Smart Contracts

Due to the fact that smart contracts are deployed on the blockchain itself, they cannot be modified after deployment. This introduces additional challenges for smart contract developers. This obstacle can partially be overcome using the `DelegateCall()`-method. When smart contract A executes this function, it is able to execute another smart contract's function (B), while the resulting changes of the function execution are persisted to smart contract A's key-value storage. Developers have been using this `DelegateCall()`-method to deploy a smart contract as proxy, with the actual implementation in another smart contract. OpenZeppelin[40] has created a plugin that makes it easy to deploy and modify upgradable smart contracts. Yet, this is not a perfect solution to the problem. If smart contracts can be upgraded by a single individual, this introduces trust and a risk of abuse. This could be overcome by requiring multiple EOAs to update the referenced contract's address before it is actually changed. When a bug has been exploited in a contract, the SCA's storage has already been changed and might not be easy to restore. Chen et al. analyzed 131 research papers on smart contracts and conducted a survey on more than 150 Ethereum developers how they maintain smart contracts to investigate these issues [22]. Chen et al. found that about 36% of the respondents upgrade their smart contracts using this proxy-approach, whereas the majority simply deployed new contracts. Some developers that deployed a new contract, also chose to destroy/invalidate the old contract using the `Selfdestruct()`-function.

[35] https://ethereum.org/en/developers/docs/smart-contracts/languages/
[36] https://soliditylang.org/
[37] https://vyper.readthedocs.io/en/latest/index.html
[38] https://github.com/ethereum/Yul-K
[39] https://github.com/fuellabs/yulp
[40] https://openzeppelin.com/

3.2.5 Ethereum Virtual Machine

Besu uses the EVM as engine for executing transactions. Wood describes the EVM as a quasi-Turing-complete because gas limits the execution time [97]. Usually, the EVM uses a `GasCalculator`[41] to keep track of the amount of gas used by the code execution. When Gas runs out, transaction execution is cancelled and the state changes are rolled back. As described in Section 3.2.1, these limitations may be lifted for a permissioned setting by configuring the gas price to 0 and by setting the gas limit of a block to the maximum value. As the gas limit approaches infinite, the EVM can be considered Turing-complete. Yet, even in a permissioned setting in which Gas is undesirable, perhaps one might still want to keep the block's gas limit reasonably low, such that a poorly written smart contract does not slow down the entire network.

By looking at the source code of Besu, it becomes apparent that the EVM-implementation[42] uses a stack-based architecture, similar to the concepts proposed in the original Ethereum Yellow Paper[97]. This EVM defines an `OperationRegistry`[43], which is a list of operations that the EVM is able to execute. These operations range from simple `Push`-[44] and `Pop`-operations[45], to more complex operations such as the `DelegateCall`-operation[46] mentioned earlier.

Transactions in Fabric are either *message calls*, which invoke a smart contract, or they are a *contract creation* transaction. We also saw in Section 3.2.3 that both types of transactions are in fact represented through the same interface. Both transactions result in a *message*, i.e. requested change to the ledger. Messages are represented in Besu by a `MessageFrame`[47], which is described as a "container object for all of the state associated with a message". In practice, this means that a `MessageFrame` has many (44) private fields including fields for the global state (`WorldState`, `blockchain`), fields for the stack architecture (`pc` for program counter, `memory`, `stack`), and much more. Describing the entire EVM-implementation of Besu would be too technical. Therefore, we summarize its architecture in Figure 3.16 and illustrate the lifecycle of executing transactions in Figure 3.17.

3.2.6 Interacting with the network

Local or remote applications can interact with the network by communicating to a single node. This communication can be performed using various protocols and APIs, as the node configuration already revealed in Section 3.2.1. The three communication techniques are JSON-RPC over HTTP, JSON-RPC over WebSockets and GraphQL over HTTP.

JSON-RPC is a standard for RPC using JSON, describing the format and rules for performing stateless communication [66]. Besu supports version 2.0 of the specification. As is common for RPC, the invoker attaches a payload to its request with the name of the method and the arguments to pass to it. An example of such a payload is given in Listing 3.5. We see each query starts with the JSON-RPC protocol version, followed by the method name, a list of arguments to pass to the parameters of the method, and finally an identifier. This identifier is merely used to tell the node it expects a response. When no identifier is given, the submitting client does not require a response. Clearly, JSON-RPC is independent of the message passing environment, and as such, it can be used in combination over both WebSockets and HTTP in Besu.

[41] https://github.com/hyperledger/besu/blob/master/ethereum/core/src/main/java/org/hyperledger/besu/ethereum/vm/GasCalculator.java

[42] https://github.com/hyperledger/besu/blob/master/ethereum/core/src/main/java/org/hyperledger/besu/ethereum/vm/EVM.java

[43] https://github.com/hyperledger/besu/blob/master/ethereum/core/src/main/java/org/hyperledger/besu/ethereum/vm/OperationRegistry.java

[44] https://github.com/hyperledger/besu/blob/master/ethereum/core/src/main/java/org/hyperledger/besu/ethereum/vm/operations/PushOperation.java

[45] https://github.com/hyperledger/besu/blob/master/ethereum/core/src/main/java/org/hyperledger/besu/ethereum/vm/operations/PopOperation.java

[46] https://github.com/hyperledger/besu/blob/master/ethereum/core/src/main/java/org/hyperledger/besu/ethereum/vm/operations/DelegateCallOperation.java

[47] https://github.com/hyperledger/besu/blob/master/ethereum/core/src/main/java/org/hyperledger/besu/ethereum/vm/MessageFrame.java

[48] https://github.com/hyperledger/besu/blob/main/ethereum/core/src/main/java/org/hyperledger/besu/ethereum/vm/MessageFrame.java#L81

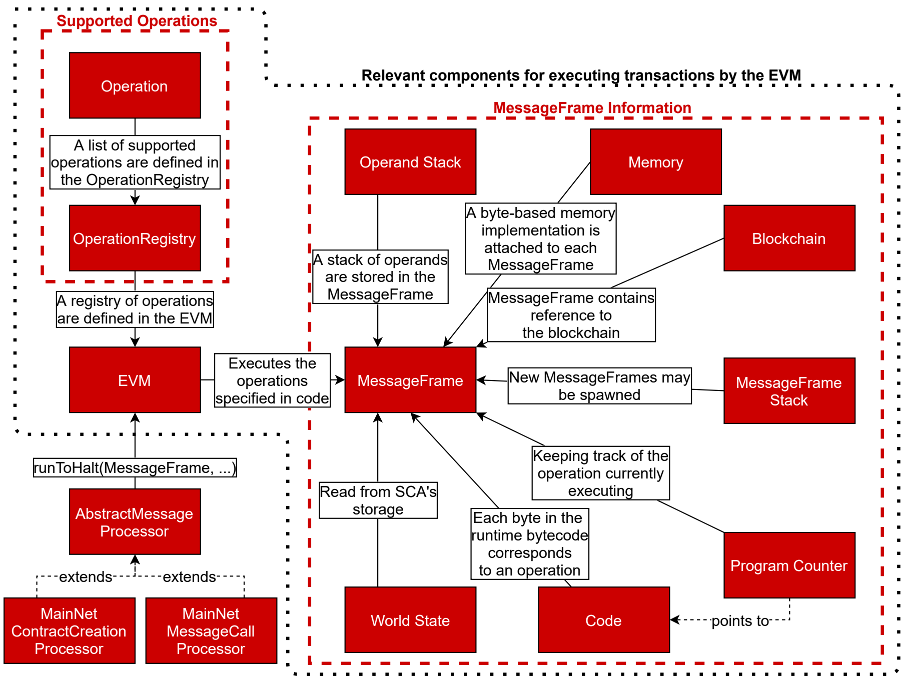

Supported Operations
Operation
A list of supported operations are defined in the OperationRegistry
OperationRegistry
A registry of operations are defined in the EVM
EVM
Executes the operations specified in code
runToHalt(MessageFrame, ...)
AbstractMessage Processor
extends
extends
MainNet ContractCreation Processor
MainNet MessageCall Processor
Relevant components for executing transactions by the EVM
MessageFrame Information
Operand Stack
Memory
A byte-based memory implementation is attached to each MessageFrame
Blockchain
A stack of operands are stored in the MessageFrame
MessageFrame contains reference to the blockchain
MessageFrame
New MessageFrames may be spawned
MessageFrame Stack
Keeping track of the operation currently executing
Read from SCA's storage
Each byte in the runtime bytecode corresponds to an operation
Program Counter
World State
Code
points to

Fig. 3.16 EVM-implementation of Besu

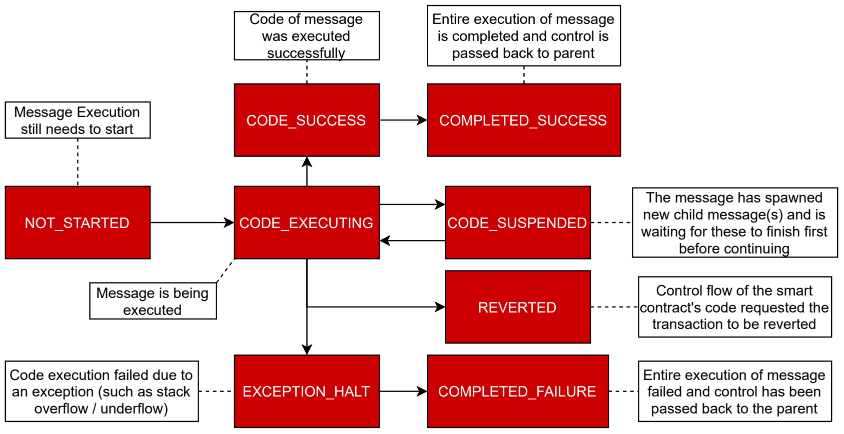

Code of message was executed successfully
Entire execution of message is completed and control is passed back to parent
Message Execution still needs to start
CODE_SUCCESS
COMPLETED_SUCCESS
NOT_STARTED
CODE_EXECUTING
CODE_SUSPENDED
The message has spawned new child message(s) and is waiting for these to finish first before continuing
Message is being executed
REVERTED
Control flow of the smart contract's code requested the transaction to be reverted
Code execution failed due to an exception (such as stack overflow / underflow)
EXCEPTION_HALT
COMPLETED_FAILURE
Entire execution of message failed and control has been passed back to the parent

Fig. 3.17 Transaction Lifecycle in Besu (derived from source code documentation at[48])

Authentication through a username-password pair or JSON Web Tokens (JWT) may be enabled by defining them on the node's configuration file, but is disabled by default.

```
{
    "jsonrpc":"2.0",
    "method":"eth_blockNumber",
    "params":[],
    "id":1
}
```

Listing 3.5 JSON-RPC example payload

The third communication technique supported by Besu is GraphQL[49] over HTTP. GraphQL is a query language in which the invoker specifies which fields to request of a certain data structure while filtering the results. Naturally, this query

[49]https://graphql.org/

language was designed to query the ledger. In addition, GraphQL may also be used to invoke other functionalities, such as to create a transaction. An example is provided in Listing 3.6, which shows how to query the number of remaining transactions.

```
{
    "query": "{pending {transactionCount}}"
}
```

Listing 3.6 GraphQL example payload

Libraries and applications

Although directly communicating with a node by constructing HTTP or WebSocket messages is valid, one could also use a library. Since Besu implements most methods of the Ethereum protocol (except sending unsigned transactions as is explained later), all libraries used to interact with the Ethereum protocol can be used for Besu. Popular libraries include Web3.js[50] for JavaScript, Web3j[51] for Java, Web3py[52] for Python, and Nethereum[53] for .Net. Besides libraries, applications for interacting with Ethereum networks can also be used for Besu, as long as they allow the specification of a custom network. Truffle[54] and Remix[55] are popular tools for deploying and testing smart contracts. Tenderly[56] could be used for gathering analytics and monitoring smart contracts. Additionally, Prometheus[57] can be used to monitor a Besu network. Grafana[58] is able to visualize the data collected by Prometheus. Figure 3.18 shows a Grafana dashboard configured for Besu. For obvious reasons, Besu-specific features (like data privacy) are unavailable when an Ethereum library is used. To use features unique to Besu, EEA-extended versions of these frameworks can be used such as Web3.js-eea[59], further explained in Section 6.2.

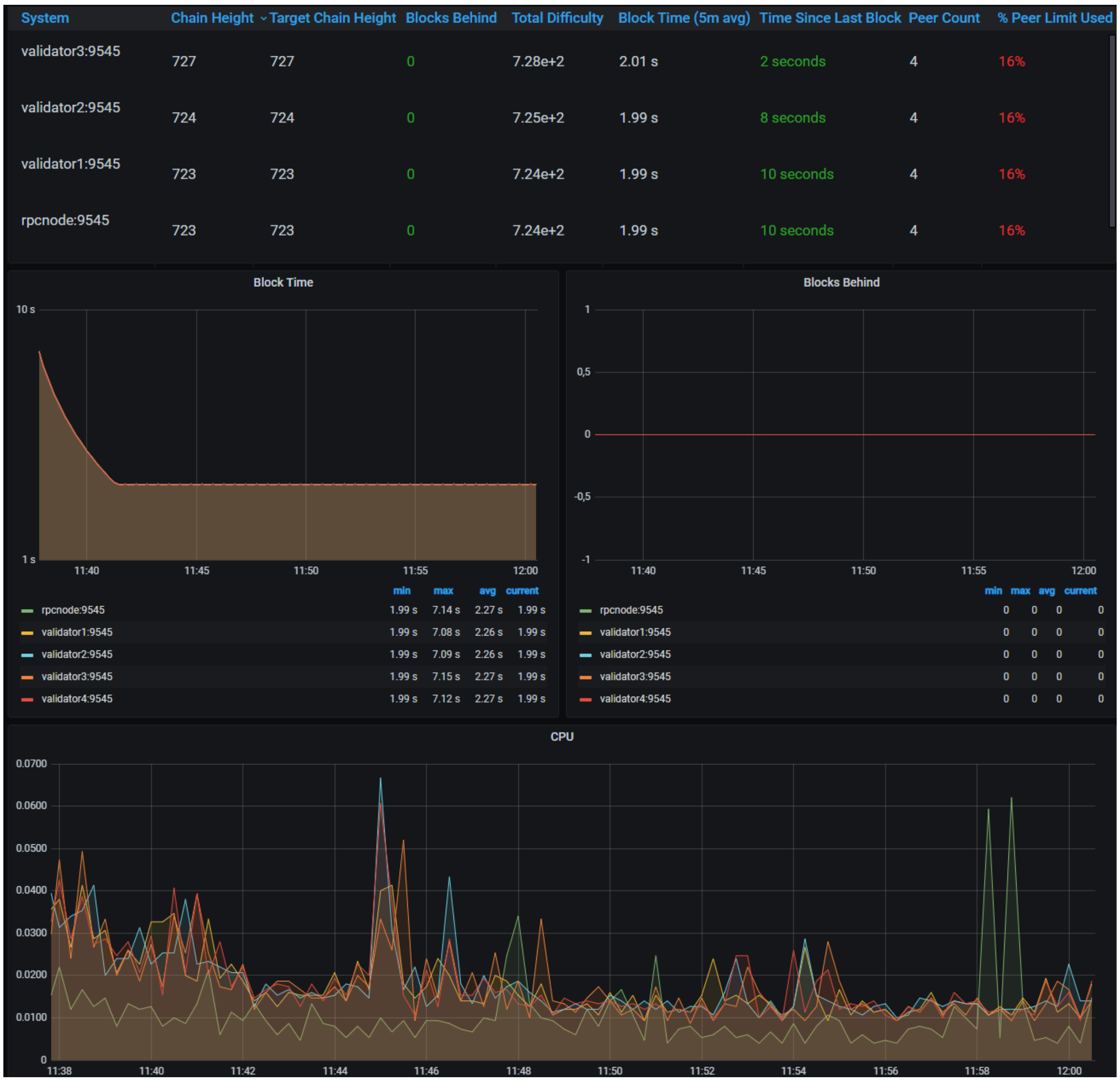
Sending transactions through EthSigner

Earlier, it was explained that transactions are not created (i.e. signed) by nodes but by accounts. An actor such as an application or user creates this transaction, and sends this transaction to a node using the JSON-RPC protocol, either directly or by using a library. For this transaction to be considered legitimate by the network, it must be signed using the account's private key. Traditional Ethereum nodes like Go Ethereum[60] and OpenEthereum[61] have an in-built wallet (i.e. account management), which stores zero or more public and private keypairs. Additionally, these Ethereum clients support the `eth_sendTransaction`-endpoint, allowing an actor to send an unsigned transaction. After receiving an unsigned transaction request, the node looks up the private key from the wallet, signs the transaction and disseminates it with the rest of the network. Besu, however, does neither support such an in-built wallet, nor does Besu support processing unsigned transactions through `eth_sendTransaction`.

Because the `eth_sendTransaction` method is unavailable to actors in Besu (as it is not implemented), the `eth_-sendRawTransaction`-method is exposed instead. Like the former, this function is also part of the Ethereum protocol. This method takes as input a signed transaction, as opposed to an unsigned one. Listing 3.7 gives an example payload for invoking the `eth_sendRawTransaction`-method using JSON-RPC.

```
{
```

[50] https://github.com/ChainSafe/web3.js
[51] https://github.com/web3j/web3j
[52] https://github.com/ethereum/web3.py
[53] https://github.com/Nethereum/Nethereum
[54] https://github.com/trufflesuite/truffle
[55] https://github.com/ethereum/remix-project
[56] https://tenderly.co/
[57] https://prometheus.io/
[58] https://grafana.com/
[59] https://github.com/ConsenSys/web3js-eea
[60] https://github.com/ethereum/go-ethereum
[61] https://github.com/openethereum/openethereum



System	Chain Height	Target Chain Height	Blocks Behind	Total Difficulty	Block Time (5m avg)	Time Since Last Block	Peer Count	% Peer Limit Used
validator3:9545	727	727	0	7.28e+2	2.01 s	2 seconds	4	16%
validator2:9545	724	724	0	7.25e+2	1.99 s	8 seconds	4	16%
validator1:9545	723	723	0	7.24e+2	1.99 s	10 seconds	4	16%
rpcnode:9545	723	723	0	7.24e+2	1.99 s	10 seconds	4	16%

Block Time

	min	max	avg	current
rpcnode:9545	1.99 s	7.14 s	2.27 s	1.99 s
validator1:9545	1.99 s	7.08 s	2.26 s	1.99 s
validator2:9545	1.99 s	7.09 s	2.26 s	1.99 s
validator3:9545	1.99 s	7.15 s	2.27 s	1.99 s
validator4:9545	1.99 s	7.12 s	2.27 s	1.99 s

Blocks Behind

	min	max	avg	current
rpcnode:9545	0	0	0	0
validator1:9545	0	0	0	0
validator2:9545	0	0	0	0
validator3:9545	0	0	0	0
validator4:9545	0	0	0	0

Fig. 3.18 Grafana Dashboard for a Besu network

```
    "jsonrpc":"2.0",
    "method":"eth_sendRawTransaction",
    "params":["0xf869018203e..."],
    "id":1
}
```

Listing 3.7 sendRawTransaction example

Using the `sendRawTransaction`-method may result in too much responsibility of applications since they all need the private key in order to create a transaction. Thus, Besu can be used with a software component called *EthSigner*[62]. Essentially, EthSigner is a proxy component in front of a node, which implements the `eth_sendTransaction`-method. Then, actors can call the `eth_sendTransaction`-method without going through all the trouble of requiring knowledge of the private key and signing the transaction. After EthSigner receives an unsigned transaction, it will sign it and forward the transaction to the `eth_sendRawTransaction`-endpoint of the node. Public and private keypairs may be stored locally

[62]https://github.com/ConsenSys/ethsigner

on the filesystem of an EthSigner instance, in a Hashicorp Vault[63], or in a Key Vault of Microsoft Azure[64]. Figure 3.19 illustrates the process of sending a transaction to a node, either directly or through EthSigner.

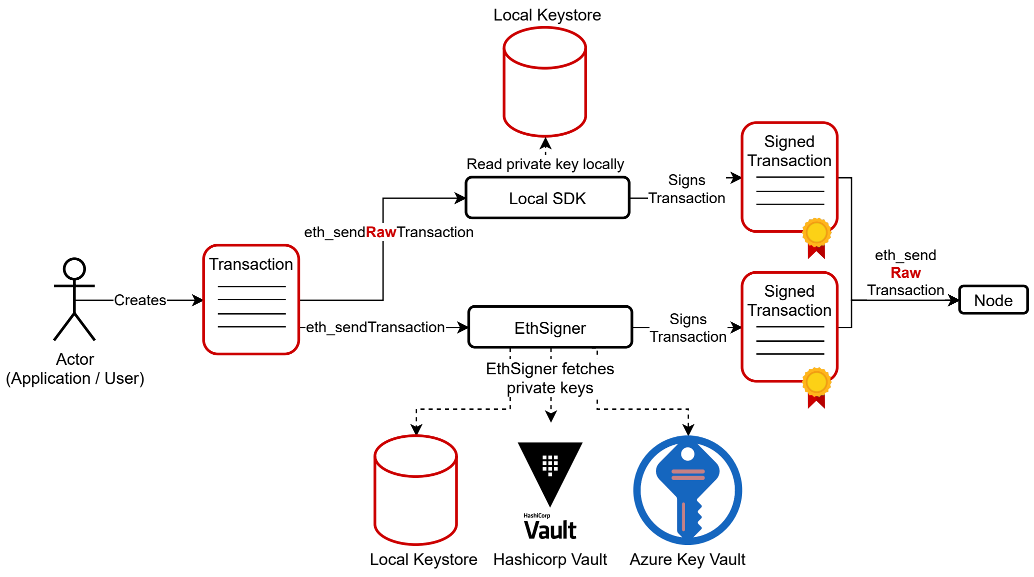


Fig. 3.19 Process of sending a transaction

3.2.7 Permissioning

Besu provides various ways to keep a network private by preventing access to non-members. Before diving into Besu's permissioning techniques, however, it is worth mentioning that vanilla Ethereum clients also have the notion of an *Ethereum Private Network*. An Ethereum private network is defined as a network not being the MainNet. Essentially, anyone could create a new Genesis block and connect a number of peers. Still, this concept of an Ethereum private network only corresponds to *isolation*, i.e. the network is unrelated to the MainNet. On the other hand, it does not imply any form of *protection* and *security*. Contrary to Ethereum private networks, Besu networks aim to provide protection and security through various permissioning techniques, explained below. Some permissioning techniques are only specified on the local node level, whereas others can be specified either locally or onchain.

Local: User-Based Authentication

Nodes can specify a `toml`-file containing a list of users allowed to invoke JSON-RPC methods. For each user, the following fields should be specified: public key, hash of a password, and a list of permissions. The permissions in this configuration file are a simple list of JSON-RPC methods a user is allowed to invoke. An example configuration is specified in Listing 3.8. This form of authentication is local and only applies to the node with users defined in its configuration file.

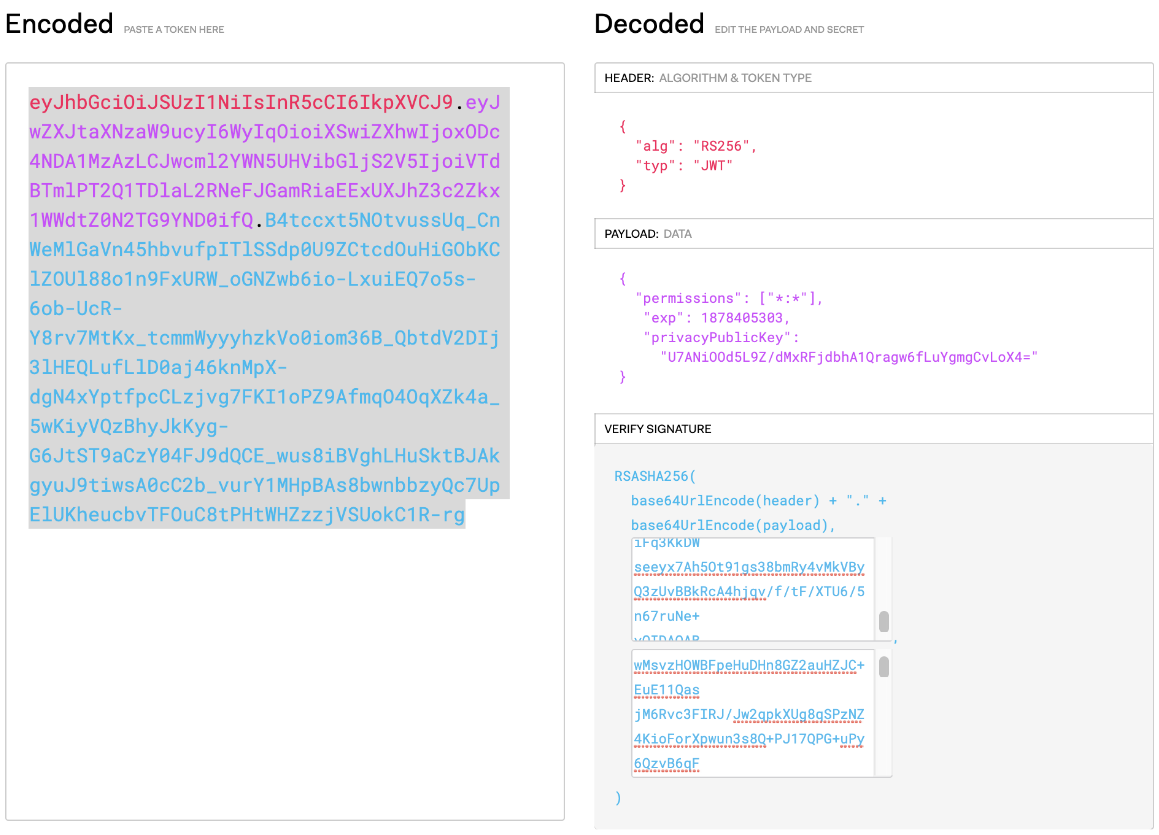
```
[Users.username1]
password = "$2a$10$l3GA7K8g6rJ/Yv.YFSygCuI9byngpEzxgWS9qEg5emYDZomQW7fGC"
permissions=["net:*","eth:blockNumber"]
privacyPublicKey="U7ANiOOd5L9Z/dMxRFjdbhA1Qragw6fLuYgmgCvLoX4="

[Users.username2]
password = "$2b$10$6sHt1J0MVUGIoNKvJiK33uaZzUwNmMmJlaVLkIwinkPiS1UBnAnF2"
permissions=["net:version","admin:*"]
privacyPublicKey="quhb1pQPGN1w8ZSZSyiIfncEAlVY/M/rauSyQ5wVMRE="
```

Listing 3.8 User-Based Authentication[65]

[63] https://www.vaultproject.io/
[64] https://azure.microsoft.com/en-us/services/key-vault/

Usernames and passwords are used to *Login* to the node by making a request. The node then creates a public and private key using RSA cryptography, which are used to generate a JWT for that user. If enabled, only clients specifying the JWT in the `Authorization` request header with valid keys are allowed to invoke functionality on the node. An example of such a token is provided in Figure 3.20.



```
eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.eyJ
wZXJtaXNzaW9ucyI6WyIqOioiXSwiZXhwIjoxODc
4NDA1MzAzLCJwcml2YWN5UHVibGljS2V5IjoiVTd
BTmlPT2Q1TDlaL2RNeFJGamRiaEExUXJhZ3c2Zkx
1WWdtZ0N2TG9YND0ifQ.B4tccxt5NOtvussUq_Cn
WeMlGaVn45hbvufpITlSSdp0U9ZCtcdOuHiGObKC
lZOUl88o1n9FxURW_oGNZwb6io-LxuiEQ7o5s-
6ob-UcR-
Y8rv7MtKx_tcmmWyyyhzkVo0iom36B_QbtdV2DIj
3lHEQLufLlD0aj46knMpX-
dgN4xYptfpcCLzjvg7FKI1oPZ9AfmqO4OqXZk4a_
5wKiyVQzBhyJkKyg-
G6JtST9aCzY04FJ9dQCE_wus8iBVghLHuSktBJAk
gyuJ9tiwsA0cC2b_vurY1MHpBAs8bwnbbzyQc7Up
ElUKheucbvTFOuC8tPHtWHZzzjVSUokC1R-rg
```

```
{
  "alg": "RS256",
  "typ": "JWT"
}
```

```
{
  "permissions": ["*:*"],
   "exp": 1878405303,
   "privacyPublicKey":
     "U7ANiOOd5L9Z/dMxRFjdbhA1Qragw6fLuYgmgCvLoX4="
}
```

```
RSASHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  iFq3KkDW
  seeyx7Ah5Ot91gs38bmRy4vMkVBy
  Q3zUvBBkRcA4hjqv/f/tF/XTU6/5
  n67ruNe+
  vOTDAQAB
  ,
  wMsvzHOWBFpeHuDHn8GZ2auHZJC+
  EuE11Qas
  jM6Rvc3FIRJ/Jw2qpkXUg8qSPzNZ
  4KioForXpwun3s8Q+PJ17QPG+uPy
  6QzvB6qF
)
```

Fig. 3.20 Private-Public Keypair Authentication using JWT[66]

Local/Network: Node Allowlist

The previous two permissioning techniques focus on actors, namely the actors that may interact with the node's API. Recall that all nodes in a Besu network link to other nodes through a so-called list of bootnodes and peer discovery, as elaborated on in Section 3.2.1. Here, we saw that transitivity applies to these connections: If node A connects to node B, and node B connects to node C, then node A also connects to node C. Moreover, an adversary might discover the address of a bootnode and connect to it, without being part of the collaborating organizations. Without restrictions, the entire network would learn of the newly connected adversary and share their data with it. A solution proposed by Besu is to have a node allowlist, which specifies a list of nodes that it may connect to. Connections from any other nodes are simply refused. This restriction is also known as the `connectionAllowed` function, as specified in the EEA Enterprise Ethereum Client specification[70].

To enable node allowlists, all nodes in the network are required to have a unique public and private key. Then, a node may be added to the node allowlist by adding its *enode*, which is a Uniform Resource Locator (URL) of the format: `enode://nodePubKey@host:port`. Each enode uniquely identifies a node in the network. This allowlist may be specified in a local `toml`-file, meaning a node with this configuration file applied will only connect to whitelisted nodes. A node that is not included in such an allowlist could still participate in the network by connecting to a different node with a different node allowlist.

[65] https://besu.hyperledger.org/en/stable/HowTo/Interact/APIs/Authentication/
[66] https://besu.hyperledger.org/en/stable/HowTo/Interact/APIs/Authentication/

Alternatively, node allowlists may be specified onchain and network-wide. Onchain permissioning works very similar to local permissioning in the sense that the application locally reads a configuration property: instead of reading from a `toml`-file, the allowlist is read from the ledger's world state. More specifically, this node allowlist is specified in a smart contract maintained by Consensys[67] which should be deployed in the Genesis block. That way, the node allowlist is automatically shared across, and applied by, all network members. Moreover, the network-wide governance structure of these configurations can be easily adapted by modifying the smart contract (1 admin can make a change, or multiple admins are required), as long as the interface does not change.

Local/Network: Account Allowlist

So far, two permissioning systems have been discussed: restricting access to a local node through a username and password, and restricting what other nodes connect to using a node allowlist. The fourth and last permissioning system to discuss is the account allowlist. Section 3.2.2 described how anyone can create one or multiple accounts offline without interacting with the network. The account allowlist discussed here aims to restrict what accounts are able to create transactions within the network, either locally or network-wide. Account allowlists are also included in the EEA Enterprise Ethereum Client specification under the `transactionAllowed` function [70].

The account allowlist is very similar to the node allowlist discussed before. Instead of having a list of enodes, account addresses are listed. Likewise, account allowlists may be specified in a local permission file or in a smart contract. Yet, there is an important difference with account allowlists that are defined onchain, as becomes clear when looking at the default implementation of the `transactionAllowed`-function in Listing 3.9. To dynamically determine if a transaction of an account is allowed, the target contract and transaction payload are also passed as argument. Thus, this function can be easily modified to define roles per account, allowing different accounts to perform different operations. And even if an account calls the same function two times with different arguments, one of the two may be rejected because of a different payload that is disallowed.

```
function transactionAllowed(
    address sender,
    address, // target
    uint256, // value
    uint256, // gasPrice
    uint256, // gasLimit
    bytes memory // payload
) public view returns (bool) {
    if (accountPermitted (sender)) {
        return true;
    } else {
        return false;
    }
}
```

Listing 3.9 transactionAllowed default implementation[68]

3.3 Findings

After having discussed both architectures of Fabric and Besu in great detail, this section provides an in-depth comparison by putting their features and aspects side-by-side. All of the architectural aspects are subdivided into a number of subsections: Section 3.3.1 describes how the concept of an organization and their identities fit into a network. Section 3.3.2 describes the structural difference of these networks, while their administration is compared in Section 3.3.3. The administration

[67]https://github.com/ConsenSys/permissioning-smart-contracts
[68]https://besu.hyperledger.org/en/stable/HowTo/Interact/APIs/Authentication/

and maintenance of individual nodes is highlighted in Section 3.3.4. Section 3.3.5 explains how the ledgers differ per framework. Differences in smart contracts are elaborated in Section 3.3.6, whereas Section 3.3.7 explains how actors interact with them in different ways.

3.3.1 Organizations and Identity

Organization

The concept of an organization is clearly defined in Fabric and has a concrete implementation in its source code. On a conceptual level, an organization forms a trusted domain in which all of its members trust each other, as was explained in Section 3.1.1. This concept is embodied by a root certificate issued by a CA. The concept of an organization does not exist in a Besu network, as all participants (i.e. accounts and nodes) are technically unrelated to each other and can only be identified through their own public key. Although the notion of an organization is unknown to the Besu framework, it could still have meaning external to the network, e.g. by associating accounts and nodes with their owners.

Digital Identity

A digital identity of an actor, i.e. an identifier of an individual or application, all represent organizations in Fabric. A digital identity is considered to act in the name of an organization, instead of on its own behalf. Moreover, these digital identities are all issued by the organizations themselves without interacting with the network. This is very powerful, since organizations have complete control over what digital identities exist without requiring approval of the rest of the network. In addition, organizations may assign each application and user its unique digital identity, or decide to issue a digital identity only for a single use to reduce the risk of being compromised. For smaller organizations, however, it could be more difficult to generate, store and disseminate these digital identities in a secure manner. As explained previously, nodes and accounts in Besu are all associated with their own unrelated public and private keypair. A direct consequence of this in a permissioned setting, is that each new node and account must be explicitly accepted by the network. This is inescapable, as the network cannot allow an unidentified and potentially malicious node to interact with other nodes, and thus have access to the ledger. Similarly, allowing any unidentified account to modify the ledger state by creating transactions should be avoided. Therefore, managing and recognizing digital identities in Besu require onchain agreement under the network participants.

In Fabric, we saw that a digital identity is represented by an X.509 certificate, whereas Besu represents an identity as a public key in plain text. No organizational hierarchy can be formed in Besu, since certificates do not exist. Both frameworks generate the public/private keypair using ECDSA.

Digital Identity Key Management

Especially in larger enterprises, it could be desirable to store credentials such as private keys separately from the applications in a secure storage facility. This can be looked at from two points of view: from the actor's perspective (user or application) and the nodes. First of all, Fabric requires actors to sign communication messages themselves, and as such, their digital identity must be locally available. While it is possible for Fabric actors to issue this identity from the Fabric CA dynamically/programmatically, an actor submitting the transaction still needs to have (indirect) access to the private key which could be a security consideration. Fabric Nodes, however, always need to have their digital identity locally defined. Similarly, Besu nodes also requires their digital identity to be defined locally, otherwise, an identity is automatically generated if none is provided. In terms of actors, Besu can be used in conjunction with EthSigner to manage credentials separately from the local actor instance, by adding a proxy for signing messages in between the node and the actor. Keys on an EthSigner instance may be stored on its local filesystem, in a Microsoft Azure Key Vault, or in a Hasicorp Vault. More information regarding EthSigner was given in Section 3.2.6.

Network Roles of Digital Identities

Commonly associated with a credential are its roles defining its permissions, i.e. the tasks that different identities may or may not perform. In Fabric, several roles can be defined in the certificate of a digital identity. Common Fabric roles are orderer, peer, client and admin. Since Fabric certificates are issued by organizations external to the network, so is the assignment of roles. Yet, the network still manages what roles of organizations are considered valid. E.g., an organization may issue an admin-role in a credential, but this admin role has no meaning without the network granting admin rights to that specific organization. In Besu, network roles do not exist in the traditional sense, although roles and their governance can be programmed in smart contracts on the application level (who can submit a transaction to contract X, what functions an account can and cannot perform). By defining these roles in this global smart contract, these roles will apply to every single smart contract deployed on the blockchain and is evaluated during transaction execution time. More on the differences between application-level roles is provided in Section 3.3.6.

Comparison Overview on Organizations and Identity

Organizations and Identity		
Concept or Feature	**Hyperledger Fabric**	**Hyperledger Besu**
Definition of Organization	Trusted domain defined as root key issued by Certificate Authority.	Unexistent, nodes and accounts are unrelated on the framework level, but could have conceptual ties externally to the framework.
Digital Identity Represents	Organizations.	Individual node or account.
Digital Identity Issued By	The organization itself, without requiring approval of the network.	Any individual, but often requires explicit approval and access by the network before valid.
Digital Identity Format	X.509 Certificate.	Plain text (public and private keys are a number, not wrapped in a certificate).
Digital Identity Signing Algorithm	Elliptic Curve Digital Signature Algorithm.	Elliptic Curve Digital Signature Algorithm.
Key Storage and Signing for Actors	Locally, but keys may be issued by application at runtime.	Locally, or on an EthSigner instance in conjunction with: its filesystem, MSA Key Vault, or Hasicorp Vault.
Key Storage and Signing for Nodes	Locally.	Locally.
Framework Network Roles	Admin, Peer, Orderer, Client.	No predefined network roles, all nodes are equal.
Network Role Issuance	Mapping of (Identity ->Role) are issued by organizations, but roles granted on the channel level.	Mapping of (Identity ->Role) are issued using any programmable governance scheme in a smart contract, but defined on the network level.

Table 3.2 Overview of differences between Fabric and Besu on Organizations and Identity

3.3.2 Network Structure

Network Composition

In terms of network composition, a Fabric network is very heterogeneous. Not only do different types of nodes exist, i.e. OSNs and peers, but there is also a difference between the peers within an organization: some peers can endorse transactions, whereas other peers cannot. The different peer types for Fabric were explained in Section 3.1.5. Furthermore, one could say that a Fabric network is logically centralized around a decentralized ordering service: without this ordering service, no blocks are created. The ordering service was explained in great detail in Section 3.1.2. Besu, on the other hand, is very homogeneous since the only type of node that really participates in the network is a full node. Although there do exist archive nodes in Besu, their functionality to the operations of the network are equivalent as those of full nodes. More information on the full node and functionality of archive nodes was given in Section 3.2.1. Because there is no reliance on a central component, any part of a Besu network might fail (up to a certain extent).

Network Isolation

In a large business network involving many parties, it is sometimes desired to isolate the network in multiple (sub)networks due to various reasons, such as to hide what subsets of organizations are actively collaborating, to protect identities of participants entirely, or to enforce disjoint ledger states. This notion of isolation is deeply embedded in the Fabric framework, since no collaborations are possible in the "main" system channel: all collaborations happen in isolated application channels. This notion of channels were explained in Section 3.1.4. In Besu, there exists only a single network. This means that without using data privacy aspects, all nodes and transactions are by default visible to all network participants. So in order to achieve isolation between groups of participants, one is required to launch and maintain completely separate Besu networks. (As we see later, data privacy can be used to hide transaction contents and ledger state, but does not result in complete isolation of participants)

Peer Discovery

As the network grows larger and more peers are connected to the network, it becomes increasingly more difficult to keep track of all of their network addresses. Thus, peer discovery is crucial to a dynamically changing network. In Besu, this is as simple as can be: each node refers *to* at least one other node as bootnode, or is referenced *by* another node. This results in an interconnected network: As long as a node is connected to the network, all nodes may be discovered by any other node. Since actors never have to interact with more than one node (a transaction can be submitted to a single node who disseminates it), peer discovery for actors is limited and often not required. In Fabric, all nodes can be discovered by other nodes within the same organization through a list of bootstrap nodes, comparable to Besu. On the inter-organizational level, however, each organization needs to hard-code the address of an anchor peer in the channel configuration. Initially, only this node can be discovered by other organizations, unless additional peers within an organization explicitly enable their discovery. For inter-organizational communication, Fabric supports peer discovery (which it calls *service discovery*) through an API-endpoint. In order for an actor to connect to another organization's node in the network, it first needs to connect to a node of its own organization, fetch a list of available nodes, select one node from that list, and connect to it. Figure 3.21 shows the difference between the two peer discovery techniques of both frameworks.

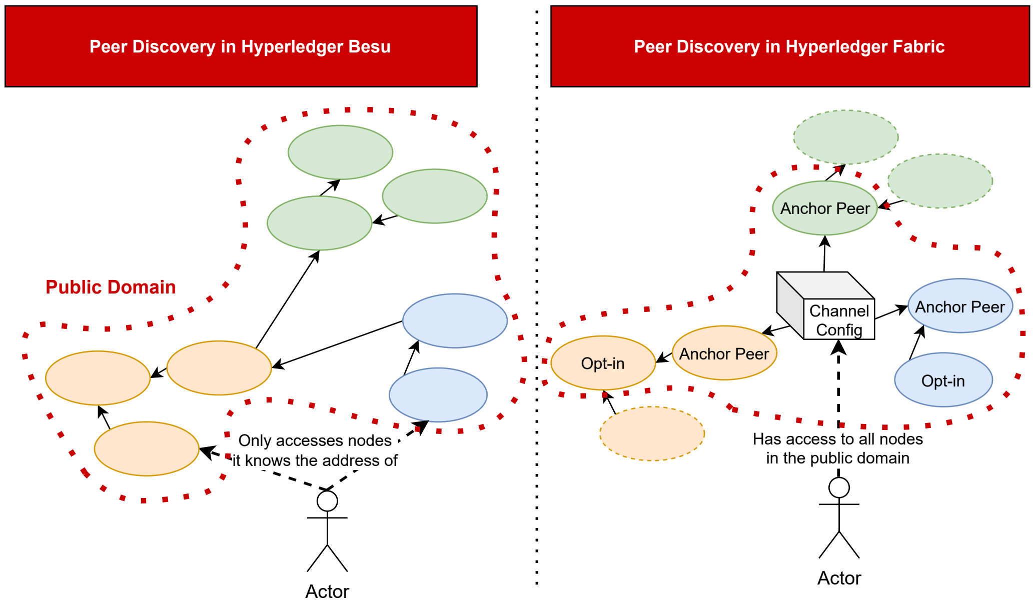

Peer Discovery in Hyperledger Besu
Public Domain
Only accesses nodes it knows the address of
Actor
Peer Discovery in Hyperledger Fabric
Anchor Peer
Channel Config
Anchor Peer
Anchor Peer
Opt-in
Opt-in
Has access to all nodes in the public domain
Actor

Fig. 3.21 Simplified view of Peer Discovery differences between Fabric and Besu

Comparison Overview on Network Structure

Network Structure		
Concept or Feature	**Hyperledger Fabric**	**Hyperledger Besu**
Network Composition	Heterogeneous (OSN, different types of peers).	Homogeneous, only similar nodes.
Network Centralization	Logically centralized around a possibly decentralized Ordering Service.	Completely decentralized, no central components.
Network Isolation	By creating channels (representing sub-networks).	By creating multiple networks.
Peer Discovery between nodes	*Within organizations*: A node may be discovered by all other nodes, as long as they are connected through bootstrap nodes. *Across organizations*: A node may be discovered by all other nodes only through opt-in.	A node may be discovered by all other nodes, as long as they are connected through bootnodes
Peer Discovery between actor-to-node	Endpoints of other nodes can be retrieved from any node, retrieved from channel configuration.	An actor can only discover neighboring nodes and not all other nodes, although often not required for most use cases.

Table 3.3 Overview of differences between Fabric and Besu on Network Structure

3.3.3 Network Administration

Network Configurability

Obviously, different business processes require different settings. Fabric is very expressive in this, in the sense that the configuration used for the genesis file consists of a lot of settings: from defining the exact certificates of each organization and their own policies, to a specification of all recognized orderers, their policies, their addresses and the addresses of the consensus nodes. As a result, significant amount of effort and expertise is required to configure a Fabric network. Although a significant amount of configurability stems from extra features and could thus be seen as advantageous, the extensive configurability also holds true for more basic settings such as connection timeouts. A Besu network's configurability, on the other hand, is very limited. To create a Besu network, one would specify the Ethereum Improvement Proposal (EIP)-version mentioned earlier, the consensus protocol and its settings in under 5 lines, and a list of smart contracts to deploy in the genesis block. Consequently, configuring a Besu network is very trivial without requiring too much expertise and effort.

Network Settings

Part of Besu's simplicity, is that the network's configuration can be specified in a single file, namely the Genesis file. When a node restarts, this Genesis file is consulted again. In Fabric, a network's configuration can be seen as the collection of a system channel's configuration and the configurations of application channels. Initially, these are specified in a `yaml`-file used to generate Genesis block(s). Consecutive updates are performed through decentralized governance by updating the channel's configuration in a transaction on the ledger. For Besu, network-wide parameters cannot be changed and are fixed once the network has started. For most settings, such as initialization nonce and initial contracts, this is not of great concern. An exception of this is the consensus protocol, which means a possibly more optimized version cannot be plugged into the network at a later point in time.

Protocol Upgrades

In Chapter 2, we saw that permissioned blockchain frameworks are actively being developed to provide more and more privacy features and performance enhancements. Therefore, upgradability of the protocol is very desirable for enterprises that start building solutions on top of these frameworks that should last for many years to come. Upgrading in Fabric makes this process very easy by having (1) backwards compatibility of nodes and (2) defining the minimally supported version within the network. If a node runs a newer version than specified in the network, those newer features are disabled.

Thus, nodes can be upgraded gradually, after which the network version may be updated to enable the new features. In Besu, nodes may upgrade to a newer Besu version as long as it implements the same Ethereum standard. Most upgrades to the Ethereum protocol are backwards compatible, meaning nodes can upgrade to the latest version without coordination. However, some of the larger upgrades to the Ethereum protocol are breaking and well-documented in EIPs[69]. To upgrade to a later version involving a breaking change, a *hard fork*[70] is required. Although it may sound like it takes a significant amount of effort to perform a hard fork, this can actually be performed very gradually as well, as long as there is sufficient coordination by the various participants. The following steps have to be taken: (1) agree on the block number to activate the new version, (2) nodes add 1 line to their genesis file specifying the new EIP at that block number, (3) nodes download the latest version and are restarted before the upgrade-block is reached, (4) when the block is reached, each node automatically enables the new features at runtime, meaning that the network is now upgraded and continuous operations as usual. To conclude, this approach is actually very similar to Fabric: first update the nodes, and then the network. The only main difference is the timing of the network upgrade: either admins update the network config in Fabric, or a block is reached in Besu. The protocol upgrade processes for both nodes are visualized in Figure 3.22.

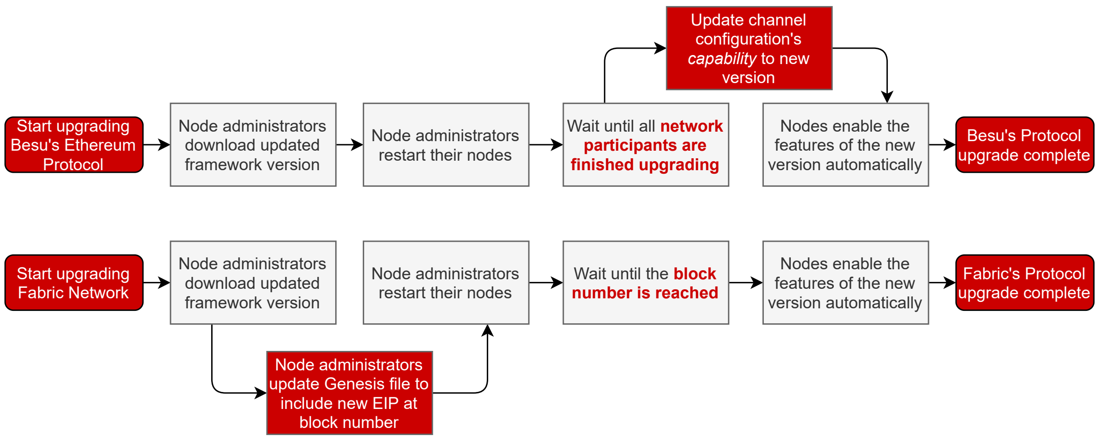

Update channel configuration's *capability* to new version
Start upgrading Besu's Ethereum Protocol
Node administrators download updated framework version
Node administrators restart their nodes
Wait until all **network participants are finished upgrading**
Nodes enable the features of the new version automatically
Besu's Protocol upgrade complete
Start upgrading Fabric Network
Node administrators download updated framework version
Node administrators restart their nodes
Wait until the **block number is reached**
Nodes enable the features of the new version automatically
Fabric's Protocol upgrade complete
Node administrators update Genesis file to include new EIP at block number

Fig. 3.22 Upgrading the blockchain protocol for Fabric and Besu

Access Management of Nodes

Another important part of network administration is network access management for only allowing identified participants to the network. For Besu, this is an optional feature, but highly recommended in a permissioned setting. Network access is managed through a smart contract deployed on the blockchain. The implementation of this smart contract is very flexible, i.e. governance may be performed by a single administrator, or a group of participants. In contrast to Besu, network access is always enabled in Fabric. A Fabric network explicitly defines all organizations in a channel configuration that are allowed to connect. The identities issued by these organizations are always allowed to connect to the network, as long as the organization (still) has access.

[69] https://eips.ethereum.org/meta
[70] A hard fork occurs when nodes are changed to accept blocks and transactions that were originally not valid on unchanged (old) nodes. The old ledger is still considered valid, but the protocol changes at some point in time.

Comparison Overview on Network Administration

Network Administration		
Concept or Feature	**Hyperledger Fabric**	**Hyperledger Besu**
Network Configurability	Very Expressive, requires high expertise and effort to configure.	Very trivial, requires not much expertise and effort to configure.
Network Settings Defined In	Collection of system channel configuration and application channels configurations located on the ledger.	A genesis file located on each of the connected nodes.
Protocol Upgrade Process	Rolling updates through channel configuration update by leveraging backwards compatibility.	Rolling updates through Hard Fork by leveraging backwards compatibility.
Protocol Upgrade performed by	Application config by Admins of app channels; Network config by Network Admins of system channel.	All individual nodes of the network.
Network Access Management of Nodes	Always enabled, a channel config defines organizations. Organizations decide externally what nodes may connect through their own identity management.	Optionally enabled, flexibly defined by software in smart contract.

Table 3.4 Overview of differences between Fabric and Besu on Network Administration

3.3.4 Node Administration

Node Configurability

Similarly to network configuration, individual nodes can also be configured differently. Node configuration is very limited in Besu and boils down to specifying which types of APIs are enabled, the hostname they listen on and the ports they expose. Additionally, each node may have unique node-level permissions defined. Fabric's configuration of nodes is again very expressive, easily having at least 200 configuration settings. These settings range from peer networking information, to parameters related to the gossip protocol like `pullInterval`, as well as various settings related to the ledger, Virtual Machine (VM), metrics, and much more.

Restricting Node Communication (Locally)

Section 3.3.3 explained the differences between Fabric and Besu with respect to network access. Additionally, it may be required to concisely define who may and may not communicate with any specific node, such as a request to read ledger state. Besu can enable username/password-based access control to generate public/private keys, as specified in the node's configuration, to only allow requests with a valid JWT. Note that these are only used to authenticate oneself with a local node, and are completely unrelated to the digital identities of nodes and accounts used by the rest of the network. A third type of access control on a local node could be to only allow the node to connect to other whitelisted nodes in the network. Similarly, a whitelist of accounts (i.e., EOA, not username/password pairs) may be maintained, and if a request is submitted by a different account, it is dropped. More details on Besu's node management was provided in Section 3.2.7. In Fabric, requests are automatically accepted if their digital identity is signed by an organization defined in the channel MSP.

Comparison Overview on Node Administration

Node Administration		
Concept or Feature	**Hyperledger Fabric**	**Hyperledger Besu**
Node Configurability	Very Expressive, requires high expertise and effort to configure.	Very trivial, requires not much expertise and effort to configure. Mostly permissions and what APIs to enable.
Node Access Management	Whitelist of connections is defined by digital identities derived from channel's organizations.	(1) Restrict actors through username/password (whitelist) using JWTs and RSA cryptography (2) Restrict the other nodes in the network to connect with (whitelist), (3) Restrict accounts that may submit a transaction (whitelist)

Table 3.5 Overview of differences between Fabric and Besu on Node Administration

3.3.5 Ledger Data

Distributed Ledger Technology and Storage Engines

The ledgers of Fabric and Besu are both fairly similar: transactions are stored in a traditional blockchain DLT for both frameworks. In Fabric, these blocks are stored as append-only files on the file system, whereas the blockchain in Besu is stored in a key-value store. Both frameworks implement a world state using a key-value store. Fabric allows the user to choose between CouchDB or LevelDB, whereas Besu uses RocksDB. Additional key-value databases may be used in combination with Besu through plugins, although one has to go through the effort of developing these first. Fabric's ledger was further explained in Section 3.1.6, whereas Besu's ledger was looked at from a technical point of view in Section 3.2.3.

Storage Encryption

Common business policy is to ensure all sensitive and possibly private data is encrypted at rest. Thus, we can look at the frameworks' support for encrypted storage. The Besu framework does not natively support encrypted storage, although it can be easily enabled by adding an encrypted storage plugin for RocksDB[71], maintained by ConsenSys. Encryption keys may be stored in Hashicorp Vault or locally. Both CouchDB and LevelDB do not support encryption at rest, and as such, is a feature unavailable to Fabric. Of course, both frameworks may still apply file-system encryption on the operating system-level.

Historic World States

Both ledgers are defined as a tuple of a transaction history and world state. This world state is maintained by applying the transactions in sequence to an initially empty memory. Obviously, the most up-to-date world state can be easily read from the ledger as long as the results of incoming transactions are applied. Retrieving historic world states, e.g. to find out the exact assets or contracts an organization owned at a certain date, may be much more difficult. One can recompute this state by applying by applying all transactions again on an empty world state up to that point in time, but this may take a very long time, proportionally to the amount of transactions in the history. To that end, archive nodes can be run on a Besu network to keep copies of the world state between all blocks, resulting in the sacrifice of storage space in favour of less computational time for historic world state. This cannot be easily achieved in Fabric without designing a software component external to the framework for storing intermediate world states.

Comparison Overview on Ledger Data

Ledger Data		
Concept or Feature	**Hyperledger Fabric**	**Hyperledger Besu**
Distributed Ledger Technology	Blockchain (append-only files), with world state (key-value store).	Blockchain (key-value store), with world state (key-value store).
Storage Engines	CouchDB/LevelDB.	RocksDB or custom plugin.
Encrypted Storage	File system encryption.	Encrypted Storage Plugin for RocksDB and File system encryption.
Reading historic world states	Unsupported, has to be recomputed from transactions which can take a long time.	Instantly available through archive nodes if any, otherwise requires expensive reconstruction from transaction history.

Table 3.6 Overview of differences between Fabric and Besu on Ledger Data

[71] https://docs.quorumplugins.consensys.net/en/stable/Concepts/Besu-Plugins/Encrypted-Storage/

3.3.6 Smart Contracts and Transactions

Smart Contract Creation

Although the concept of a smart contract as a distributed application is the same for both Fabric and Besu, their implementations are fundamentally different. In Besu, any network participant may generally create a smart contract. Then, this smart contract is shared with other nodes by storing it on the ledger. Since the actual code of the smart contract is stored on the ledger through contract creation transactions, all nodes in the network have the exact same smart contract code. Locally changing this smart contract is useless, since that will result in a fork of the blockchain. In Fabric, it can be configured in each channel's configuration who may deploy a smart contract. This is set through policies and may be, for instance, a single administrator, any participant, or a majority of channel administrators. This deployment, however, is not similar in Fabric compared to Besu. A deployed smart contract in Fabric consists merely of a name and version identifier, rather than the actual code as is the case in Besu. This code actually has to be deployed separately on every single node by an administrator. Since the code is unrelated to the deployment on the blockchain, any node is able to deploy slightly different adaptations of the chaincode. To exemplify this, an organization may (ab)use this to only endorse a transaction if it has a personal or financial interest in its results.

Upgrading Smart Contracts

In Fabric, upgrading a smart contract is very similar to the creation of a new smart contract. First, administrators install the updated chaincode on all nodes. After that, the actual chaincode that is used network-wide is updated by incrementing the version-field of the chaincode trace on the ledger. This updating process is governed through the same type of policy as its creation, set to a majority of administrators by default. In Besu, updating a smart contract is very nontrivial in the sense that all smart contracts are final. Thus, to "upgrade" a smart contract, one might deploy a new version on the ledger and import the old state. This can be a very complicated process. Alternatively, developers may deploy 2 contracts: the smart contract containing business logic and a proxy smart contract deferring to it. Then, the proxy can be invoked to execute transactions, whereas the state is still stored in the proxy contract, rather than the contract being invoked. To upgrade the business logic, one deploys a new contract and points the proxy to the newly created instance.

Smart Contract Languages

Smart contracts can be written using a variety of languages for both frameworks. Yet, there is an important distinction. Fabric aims for wide adoption by supporting general-purpose programming languages including Golang, Java and Node.js. For each of these general-purpose programming languages, interfaces[72] are used to write contract functionality. In contrary, Besu's programming languages are uniquely designed for smart contracts, as opposed to any task, i.e. DSLs. In essence, any programming language for the Ethereum protocol can be used for Besu. Currently, this includes Solidity, Vyper and Yul(+). Besides the programming languages themselves, the approach to writing smart contracts and how one manipulates the ledger is very different as well. In Fabric, the ledger's key-value store is directly being manipulated through `Get`, `Put`, and `Delete`-operations. Although this also happens under the hood with Besu's Solidity and Vyper, the actual code being written is on a much higher level of abstraction. To give a concrete example, adding a value to a `mapping` in Besu is achieved through code such as `myMap[addr] = 123`, which indirectly maps to some key in the key-value space. Mappings in Fabric are unsupported as an operation, and either requires an entire list to be deserialized, modified and serialized (which is very inefficient!), or by having an application-level mapping of an entry to a specific key in the key-value's storage space.

[72]For Golang which does not have interfaces, this works slightly differently by returning a pointer to a smart contract struct in each function.

Intranetwork Composability

Composability is a powerful attribute of smart contracts, meaning a smart contract can be deployed and used independently of others, while still being able to invoke other smart contracts. First, let's look at composability within a single Fabric channel or Besu network. In Besu, any smart contract can call any other smart contract for both read and write operations. Intrachannel composability in Fabric is very similar: A parent transaction execution halts until the execution of the `invokeChaincode`-operation is completed. The results of calling another chaincode are appended to the readset and writeset of the overarching parent call. Yet, there is a subtle limitation here. Recall that a chaincode is installed locally on a node. Thus, composability with another chaincode is only supported if it is also installed on that same node. Any node in the network without all chaincodes of the callgraph installed, is unable to successfully endorse a transaction.

Internetwork Composability

Composability may not only exist within the same network, but also between multiple channels and networks, i.e. internetwork composability. In Besu, this is not supported by the framework, nor by the EEA Enterprise Ethereum Client specification. Yet, this specification states that "future work [...] is expected to include [...] possible incorporation of work from EEA's Cross-chain interoperability Group" [70]. In Fabric, inter-channel communication is only allowed by performing local read operations on another channel's ledger. Transactions across multiple Fabric networks is also not a feature embedded in the framework itself.

Despite this, a significant amount of research has been published on cross-chain communication and transactions. For instance, Belchior et al. published a survey in which they analyzed 332 documents on blockchain interoperability [13]. This survey includes mechanisms for fungible assets (i.e. cryptocurrencies) such as by having *two-way pegs* for sidechains [84], *notary schemes* as third-party software components mediating between blockchains [19], and *cross-chain atomic transactions* using *Hashed Time-Locks Contracts* [46]. These solutions, however, may only work on similar blockchain frameworks and not across platforms. A second category identified by Belchior et al., comprises the introduction of a coordination blockchain, called *blockchain engines* [13]. Essentially, they create a shared and layered infrastructure by observing blocks in multiple other blockchains and linking them. In the permissionless sector, this type of blockchain coordination is applied by *Polkadot* using *Parachains* [96] and *Cosmos* using *Zones* [55]. Most importantly, however, for permissioned blockchains is the category of *Blockchain Connectors*. This category comprises *Blockchain-Agnostic Protocols* by considering a blockchain like an *Autonomous System* in network engineering, by essentially focusing on the routing and networking aspect of messages between two blockchains [42, 52]. For instance, *Hyperledger Quilt*[73] implements one such *Interledger Protocol* [92]. The category of blockchain connectors also comprise the sub-category of *Trusted Relays*, which are trusted parties to redirect transactions between multiple blockchains. *Hyperledger Cactus*[74] is such a trusted relay similar to the previously mentioned notary scheme [64]. The trusted third party of Hyperledger Cactus may be a decentralized network in itself and supports escrowed asset transfers. At the time of writing, Hyperledger Cactus supports ledger operations across Besu, Fabric, R3 Corda and Quorum. Thus, even though Fabric and Besu both do not support operations across multiple isolated networks (yet), this is a very active field of research. Moreover, enterprise solutions such as Hyperledger Cactus are starting to emerge, allowing cross-channel operations without requiring support for these operations by the frameworks themselves.

Submitting Transactions

Different types of entities invoke smart contracts in both frameworks. In Besu, a transaction is submitted by an *account*, which is a self-created public/private keypair that may or may not be associated with a particular organization. In Fabric, a transaction is submitted by an actor explicitly given the *client* role by the organization through its digital identity. An additional layer of security could be to have permission-like restrictions in place that only allow certain transactions to be

[73]https://www.hyperledger.org/use/quilt
[74]https://www.hyperledger.org/use/cactus

submitted by a subset of actors. This is easy to achieve in Besu: a particular permissions smart contract can be installed on the ledger. This contract determines if certain transactions are allowed by looking at the submitting account, invoked smart contract, and the transaction's payload (`transactionAllowed`). Of course, the same can be achieved in each individual smart contract as well, although if there are many contracts, defining the roles of all participants for each contract could be cumbersome. In Fabric, the same functionality is slightly more difficult to achieve. Most straight-forward would be to implement access control in the chaincode by fetching the identity of the client submitting the transaction proposal. This may be used in conjunction with Attribute-Based Access Control (ABAC) derived from the client's identity. Yet, this is only a local restriction on a node and requires this restriction to be put in a larger context for enforcement. To enforce this restriction network-wide, a secure endorsement policy should be used together with the aforementioned identity restrictions in the chaincode locally.

Comparison Overview on Smart Contracts and Transactions

Smart Contracts and Transactions		
Concept or Feature	**Hyperledger Fabric**	**Hyperledger Besu**
Smart Contract Installation Process	Manually by local node administrators.	Automatically through transactions.
Smart Contract Installation Location	Locally on each node, with a trace on network.	Smart Contract's code is stored on the ledger.
Smart Contract Consistency	Implementation may differ per node.	Always the same across nodes, otherwise the deviating implementation's node creates a hard fork.
Smart Contract Approval	Configurable in each channel, majority approval by default.	Not required by default, but the creation transaction may be restricted on a software level.
Upgrading Smart Contracts	Node Admins install new version, organizations approve changes on the ledger.	Deploy new contract or update the reference of a proxy contract in front of the core business logic.
Smart Contract Languages	General-purpose programming language (Golang, Java, Node.js).	DSL (Solidity, Vyper, Yul, Yul+).
World State Modification Approach	Direct modification of key-values using Get/Put/Delete operations.	Indirect manipulation using high-level language like Solidity, or lower level language like Yul.
Intranetwork Composability	Read+Write, but only if both chaincodes installed.	Read+Write.
Internetwork composability	Between channels: only local read operations are supported. Between networks: no framework support, but Blockchain connectors external to the framework may be used, such as Hyperledger Cactus as Trusted Relay.	No framework support, although the EEA has a working group to research this. Blockchain connectors external to the framework may be used, such as Hyperledger Cactus as Trusted Relay.
Transaction Submitter Identity	Digital identities of organizations with the Client role.	Accounts.
Transaction-based restrictions	Restricting who may invoke particular functions should be defined in chaincode, optionally in conjunction with ABAC, but always with a strong and restrictive (chaincode-level or state-based) endorsement policy.	In the smart contracts containing the business logic, or in a transactions permissions contract that applies to all smart contracts.

Table 3.7 Overview of differences between Fabric and Besu on Smart Contracts and Transactions

3.3.7 Actor-Node Interaction

Actors have multiple ways of interacting with a network through a node in both frameworks. In terms of directly communicating over a messaging protocol, an actor can send requests directly to Fabric using the gRPC-protocol. One should be aware its documentation is limited, though. For Besu, an actor may interact with a node in a variety of ways: JSON-RPC over HTTP, JSON-RPC over WebSockets, or GraphQL over HTTP. In addition to this, both frameworks provide out-of-the-box libraries to make this interaction easier. Fabric provides these libraries in the form of SDKs for Java, Node.js. Python and Golang SDKs are still in active development and experimental. For Besu, any library that can be used with the Ethereum protocol works with Besu, such as Web3.js for JavaScript, Web3j for Java, Web3py for Python and Nethereum for .Net.

Comparison Overview on Actor-Node Interactions

Actor-Node Interaction		
Concept or Feature	**Hyperledger Fabric**	**Hyperledger Besu**
APIs exposed by node	gRPC.	JSON-RPC over HTTP, JSON-RPC over WebSockets, GraphQL over HTTP.
Supported Libraries for Actors	SDKs for Java, Node.js. Golang and Python are in active development.	Any Ethereum libraries may be used. These include Web3.js for Java; Web3j for Java; Web3py for Python; Nethereum for .Net.

Table 3.8 Overview of differences between Fabric and Besu on Actor-Node Interaction

Chapter 4

Transaction Flow

A *transaction flow* denotes the process an actor follows from creating a transaction (proposal) to the commit of the transaction to the ledger, resulting in the ledger's world state being updated. This chapter concerns itself with collecting knowledge to answer the research question: "***What is the transaction flow of Besu and Fabric from commit to validation?***" To answer this question, the transaction flow of both frameworks is analyzed in great detail. Again, this section assumes the public transaction flow without considering data privacy, as this is expanded on in Section 6.

4.1 Transactions in Fabric

Fabric's transaction flow takes an entirely novel approach compared to the literature and other DLT frameworks. Namely, transactions are executed before they are ordered into blocks by the ordering service. Then, the ordering service disseminates the blocks to the nodes of a channel, which all validate and commit each individual transaction. Due to the order in which these steps take place, Fabric's architecture is commonly referred to as an *execute-order-validate architecture* [3]. We analyze the transaction flow of Fabric by predominantly using Fabric's source code as source of truth, in order to ensure that the information is correct, up-to-date and not based on an older version of Fabric. The transaction flow is investigated using the `asset-transfer-basic` smart contract from the samples GitHub repository[1], using a Java application interacting with it[2]. This Java application communicates with a node using the Java SDK[3].

The remainder of this section is split in the following (chronological) parts: First, the creation of a transaction proposal and the forwarding of it to endorsers is explained in Section 4.1.1. Then, Section 4.1.2 elaborates on the endorsing process to generate the changes that should be applied to the ledger. After all endorsements are collected by the actor, the endorsements are validated (Section 4.1.3) and broadcasted to the ordering service (Section 4.1.4). After having received a transaction, the ordering service processes the request (Section 4.1.5) and is ordered into blocks through a consensus protocol (Section 5.1). This block is disseminated to the peers of a channel (Section 4.1.6) with the help of a Gossip Protocol. Finally, each node validates the transactions included in the block (Section 4.1.7) and commits the changes to its ledger (Section 4.1.8). This high-level transaction flow is visualized in Figure 4.1. As can be seen from the image, various phases or stages can be identified. Each of these stages, from the start of the flow to its end, are dissected to its exact steps. The interested reader may compare this flow diagram with a (less-detailed) sequence diagram provided by the Fabric documentation in Appendix A.2.

[1]https://github.com/hyperledger/fabric-samples/tree/master/asset-transfer-basic/chaincode-java
[2]https://github.com/hyperledger/fabric-samples/tree/master/asset-transfer-basic/application-java
[3]https://github.com/hyperledger/fabric-gateway-java

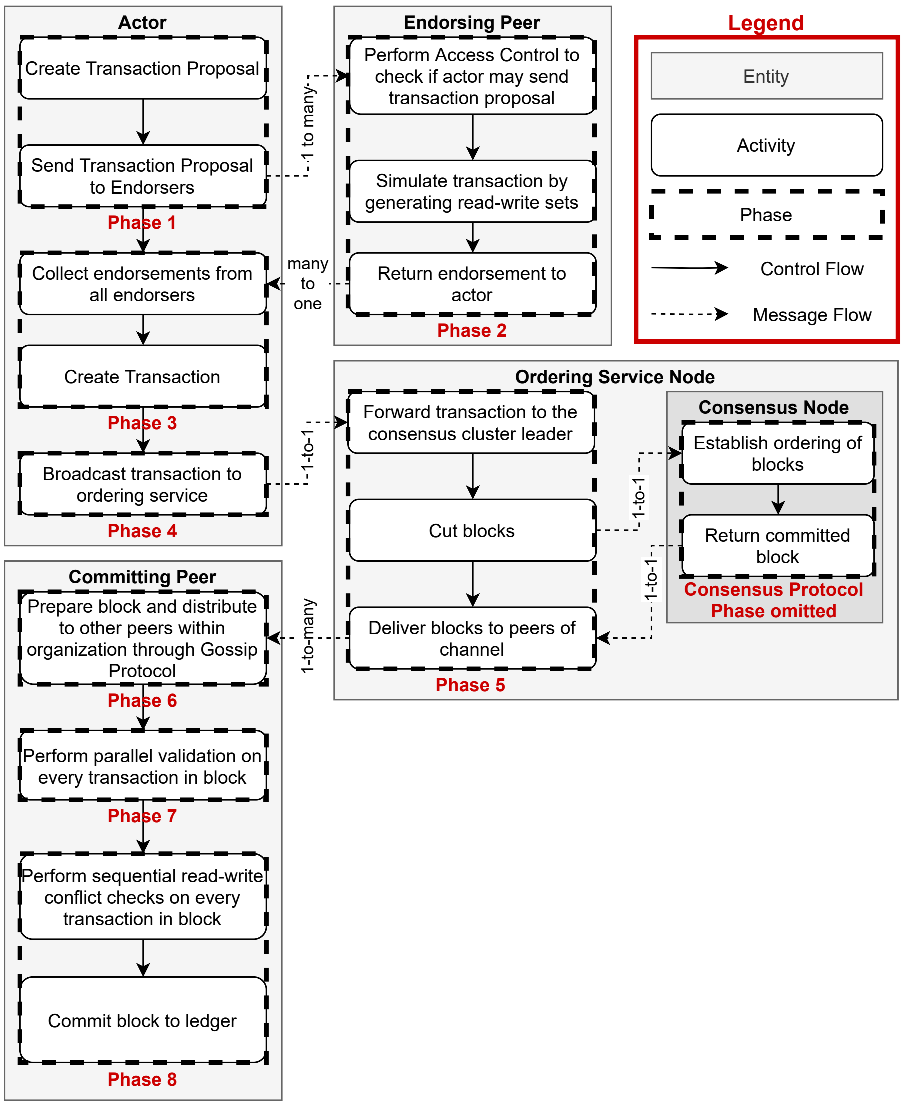

Actor
Create Transaction Proposal
Send Transaction Proposal to Endorsers
Phase 1
Collect endorsements from all endorsers
Create Transaction
Phase 3
Broadcast transaction to ordering service
Phase 4
Endorsing Peer
Perform Access Control to check if actor may send transaction proposal
Simulate transaction by generating read-write sets
Return endorsement to actor
Phase 2
1 to many
many to one
Legend
Entity
Activity
Phase
Control Flow
Message Flow
Ordering Service Node
Forward transaction to the consensus cluster leader
Cut blocks
Deliver blocks to peers of channel
Phase 5
1-to-1
Consensus Node
Establish ordering of blocks
Return committed block
Consensus Protocol Phase omitted
1-to-1
1-to-1
Committing Peer
Prepare block and distribute to other peers within organization through Gossip Protocol
Phase 6
1-to-many
Perform parallel validation on every transaction in block
Phase 7
Perform sequential read-write conflict checks on every transaction in block
Commit block to ledger
Phase 8

Fig. 4.1 High-level overview of Fabric's Transaction Flow

4.1.1 Phase 1: Setting up the Transaction Proposal

The first step of the transaction flow has to be performed by an actor, namely to create a transaction (proposal). Consider the case in which the application wishes to create an asset through the ledger. In terms of application code, only a couple of steps need to be taken. First, the application needs to have its digital identity ready, or issue a new identity from a CA. This digital identity is used to connect to the network's gateway using its identity. Although the name might suggest otherwise, this gateway is not really a network component, but rather a library to manage connections. The actor tells the gateway what channel to connect to and the smart contract using its label, as specified during the contract creation (Section 3.1.7). Then, the actor can create a transaction by simply calling the `submitTransaction` function on the contract. A simplified view on how to submit a transaction is provided in Listing 4.1. We can see that to invoke a function, the actor only needs to pass a function name and its arguments.

```
public class App {
    public static Gateway connect() {
        // Wallet is a file with the digital identity
        Gateway.Builder builder = Gateway.createBuilder();
        builder.identity(wallet, "appUser").networkConfig(networkConfigPath).discovery(true);
```

```java
        return builder.connect();
    }

    public static void main(String[] args) {
        Gateway gateway = connect();
        Network network = gateway.getNetwork("mychannel");
        Contract contract = network.getContract("basic");
        contract.submitTransaction("CreateAsset", "asset13", "yellow", "5", "Tom", "1300");
    }

}
```

Listing 4.1 Submitting a transaction from an application

In fact, even though the application is not concerned with the creation of a `Transaction`-instance and populating its fields, this is actually automatically performed by the SDK. Calling the `submitTransaction`, as seen in Listing 4.1, results in three steps performed by the SDK. First, the transaction is endorsed by fetching `ProposalResponses`. Secondly, these responses are validated. Lastly, the transaction is committed to the ledger. Listing 4.2 shows what this `submit` code looks, and concretizes these steps for the interested reader.

```java
public final class TransactionImpl implements Transaction {
    public byte[] submit(final String... args) throws Exception {
        Collection<ProposalResponse> proposalResponses = endorseTransaction(args);
        Collection<ProposalResponse> validResponses = validatePeerResponses(proposalResponses)
    ;
        return commitTransaction(validResponses);
    }
}
```

Listing 4.2 The submission of a transaction by the SDK

There are mainly two different techniques for collecting the endorsements, depending on the arguments. In case the application does not specify any endorsers, as is the case in our example, the SDK will first make a request to a node in order to find out the chaincode's callgraph. This callgraph includes all other chaincodes that are invoked by the proposal's chaincode. After the actor has received a response, it now has a list of chaincodes that a committer node must have installed before it is technically able to endorse a transaction.

Now, for each of the individual chaincodes in the callgraph, a request is made to obtain their chaincode details on the channel's ledger. The application extracts the (chaincode-level) endorsement policy for each of these chaincodes. While this endorsement policy only contains the organizational *roles* that must endorse a transaction, the application substitutes each identifier by a set of node *endpoints* discovered through the channel's anchor nodes, e.g. `Org1.peer` is replaced by `127.0.0.1:987`. Then, a filter is applied to only include endpoints of nodes that are marked as endorser. At this point, the application has discovered (1) what chaincodes will be (indirectly) called, (2) what their endorsement policies are (using a nested structure), and (3) what nodes can technically endorse each chaincode. The only step that remains now is to randomly select a set of endorsers having all required chaincodes installed, which can together satisfy the endorsement policy.

Recall that the set of endorsers did not *need* to be discovered, but could also be provided as a set of endpoints by the user. Either way, we now have a chosen set of endorsers and control flow continues in the same manner. The application sends the transaction proposal to all endorser endpoints. The flow of this phase is summarized in Figure 4.2.

4.1.2 Phase 2: The Endorsement Process

After an endorser receives a request to endorse a transaction proposal, it unpacks the proposal and generates a hash. The proposal is first validated by checking all required fields are non-empty. Moreover, the validator ensures that the digital

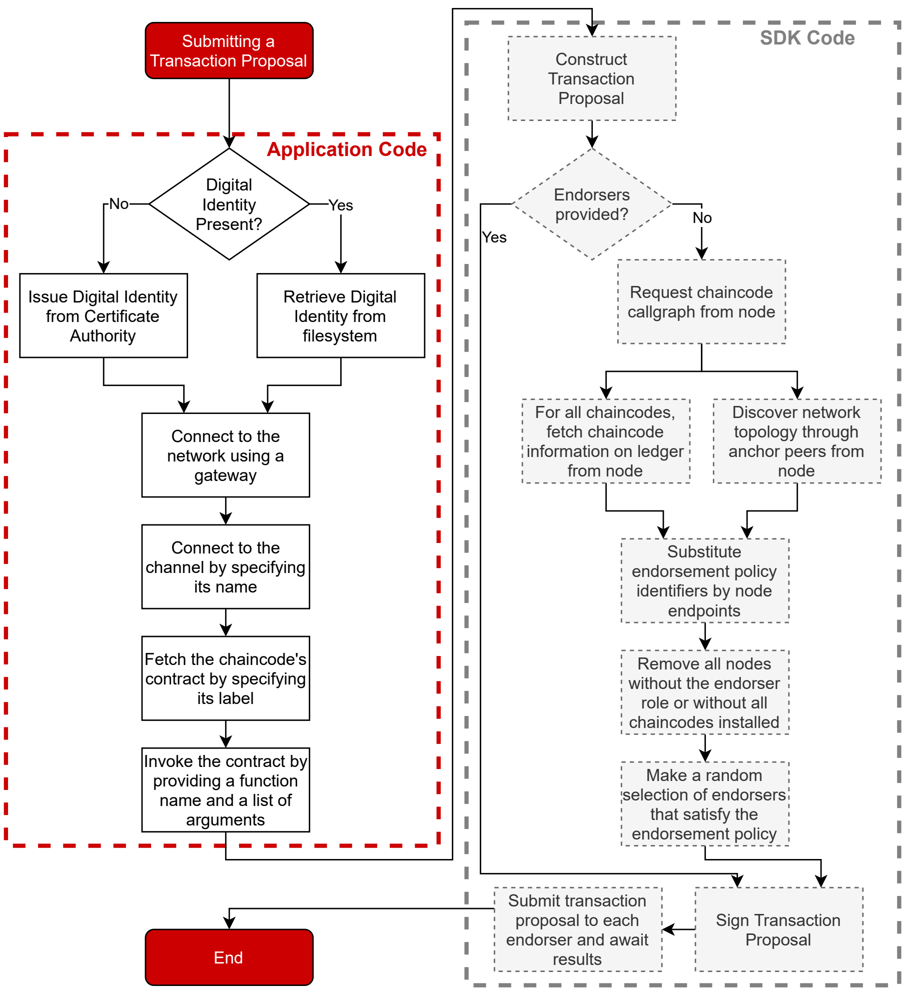

Submitting a Transaction Proposal
Application Code
Digital Identity Present?
No
Yes
Issue Digital Identity from Certificate Authority
Retrieve Digital Identity from filesystem
Connect to the network using a gateway
Connect to the channel by specifying its name
Fetch the chaincode's contract by specifying its label
Invoke the contract by providing a function name and a list of arguments
SDK Code
Construct Transaction Proposal
Endorsers provided?
Yes
No
Request chaincode callgraph from node
For all chaincodes, fetch chaincode information on ledger from node
Discover network topology through anchor peers from node
Substitute endorsement policy identifiers by node endpoints
Remove all nodes without the endorser role or without all chaincodes installed
Make a random selection of endorsers that satisfy the endorsement policy
Sign Transaction Proposal
Submit transaction proposal to each endorser and await results
End

Fig. 4.2 The creation and sending of a transaction proposal

identity of the proposal's creator is properly defined, exists in the channel's MSP, and has access according to the channel's policy. Furthermore, the validator ensures the signature is valid. If the transaction is a duplicate, it is discarded as well.

To endorse the transaction, first a *Simulator* is being created. This Simulator is a wrapper for the ledger and exposes two types of functions: functions to update and query the ledger, as well as a function to fetch the simulation results once finished. Internally, the simulator maintains a read-write set, as explained in Section 3.1.6. Thus, by using this wrapper, every single read and write request of the ledger can be tracked and recorded. Moreover, a request to update the ledger's state does not actually modify the ledger itself, but these changes are remembered in memory. This prevents the peer node from having to undo the changes afterwards, since changes are never made to the ledger.

The actual execution of the simulation is up next. Just before the start of this simulation, a shared lock is created over the entire ledger, meaning no ledger updates can be performed while the transaction is being simulated. Then, the chaincode on the endorser's file system is invoked, using the provided function name and arguments. If the chaincode execution takes longer than defined in the local node's configuration file, the execution is aborted. After this execution, the read- and writeset are returned from the simulator. The simulation results are being marshalled, and the ledger's lock is released. Whether the endorsement call has failed, timed out, or succeeded, the results are returned to the actor requesting the invocation. This endorsement process is illustrated in Figure 4.3 and is often referred to as Endorsement System Chaincode (ESCC).

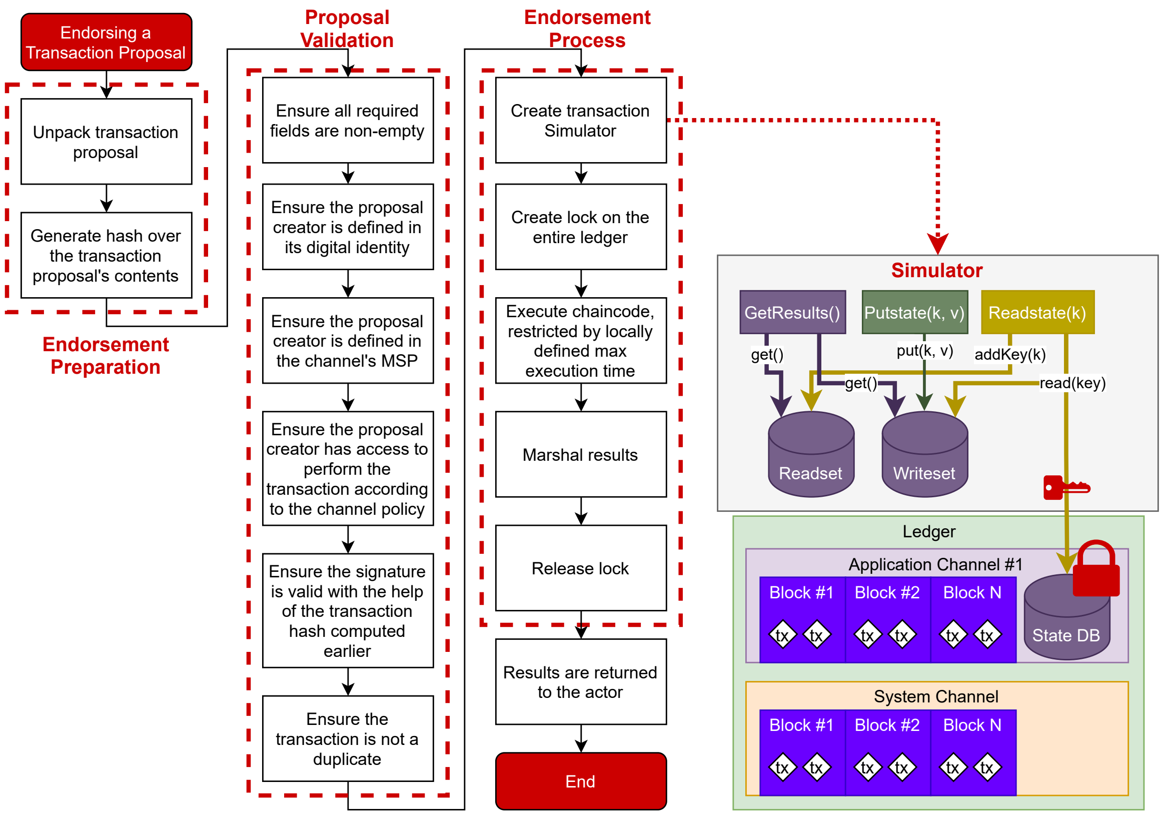

Endorsing a Transaction Proposal
Unpack transaction proposal
Generate hash over the transaction proposal's contents
Endorsement Preparation
Proposal Validation
Ensure all required fields are non-empty
Ensure the proposal creator is defined in its digital identity
Ensure the proposal creator is defined in the channel's MSP
Ensure the proposal creator has access to perform the transaction according to the channel policy
Ensure the signature is valid with the help of the transaction hash computed earlier
Ensure the transaction is not a duplicate
Endorsement Process
Create transaction Simulator
Create lock on the entire ledger
Execute chaincode, restricted by locally defined max execution time
Marshal results
Release lock
Results are returned to the actor
End
Simulator
GetResults()
Putstate(k, v)
Readstate(k)
get()
put(k, v)
addKey(k)
get()
read(key)
Readset
Writeset
Ledger
Application Channel #1
Block #1
Block #2
Block N
tx
State DB
System Channel

Fig. 4.3 The endorsement process

4.1.3 Phase 3: Validation of endorsements

After all endorsers have returned their endorsements, control flow is back at the actor (or, for our current example, the application's gateway). The SDK creates a *ProposalResponse*, which consists of the original transaction proposal of the client itself, the endorser's response, and the peer to which the proposal was sent. The actor then verifies whether the response was valid. In the previous phase, it was explained that invalid results could be returned. Example situations in which invalid results are returned, is when the chaincode invocation failed or when the simulation took too long and was aborted. If the signature is missing, or if the signature is invalid (does not match the endorser's certificate), the ProposalResponse is marked as invalid and discarded. Figure 4.4 illustrates this relatively trivial phase, as well as the most important fields of the ProposalResponse's structure. The reader should note that the actual ProposalResponse *object* contains a ProposalResponse-*field*. The ProposalResponse's *field* is used to store the endorsements themselves, which remain signed by the endorser.

4.1.4 Phase 4: Broadcasting the Transaction

The previous phase of *validating an endorsement* was performed for all individual responses. Thus, the actor now has a list of valid ProposalResponses. Before these ProposalResponses are broadcast to the ordering service, the actor needs to decide the orderer to connect to. The actor picks an orderer by reading all orderers from its local configuration file and its user context (containing its digital identity, among others). To ensure that transactions are not always submitted to the same orderer (in case multiple exist and are specified), the list of known orderers is optionally being shuffled.

Up until this point, all proposal responses (i.e. endorsements) have only been verified individually, without ensuring that their results are consistent with each other. In fact, the readset and writeset of all responses need to be the same, otherwise a transaction would be rejected during a later validation phase. Optionally, the application may perform this consistency

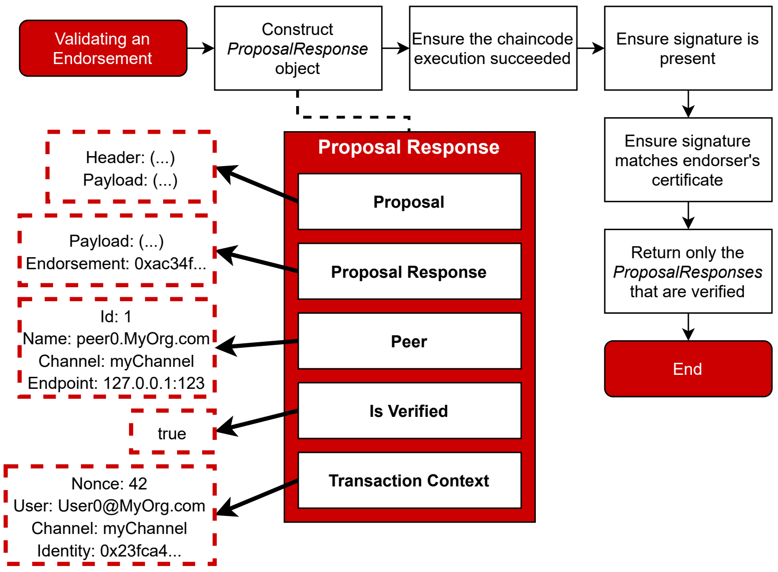


Fig. 4.4 Validating Proposal Responses

check or skip it, although skipping it could mean the transaction could be invalidated by the network during a later phase. For no obvious reason, this consistency check is only performed now, at the phase in which results are (prepared for being) sent to the orderer, rather than during the validation phase. If any of the endorsements are inconsistent, the entire transaction flow is aborted. If this consistency check succeeds, the transaction proposal responses are merged into a single *transaction*. The exact structure of such a transaction was discussed in Section 3.1.6 and illustrated in Figure 3.8.

Finally, this transaction is sent to the first orderer in the list of shuffled orderers. If this request is successful, it means the transaction is now waiting to be included in a block by one of the OSNs. In case the request fails, the SDK tries to send the transaction to the next orderer, until no orderers are left. In both scenarios in which either the transaction is received by *exactly one* orderer, or the transaction could not be sent to *any* of the orderers, the SDK finishes its execution. Again, this flow is illustrated in Figure 4.5.

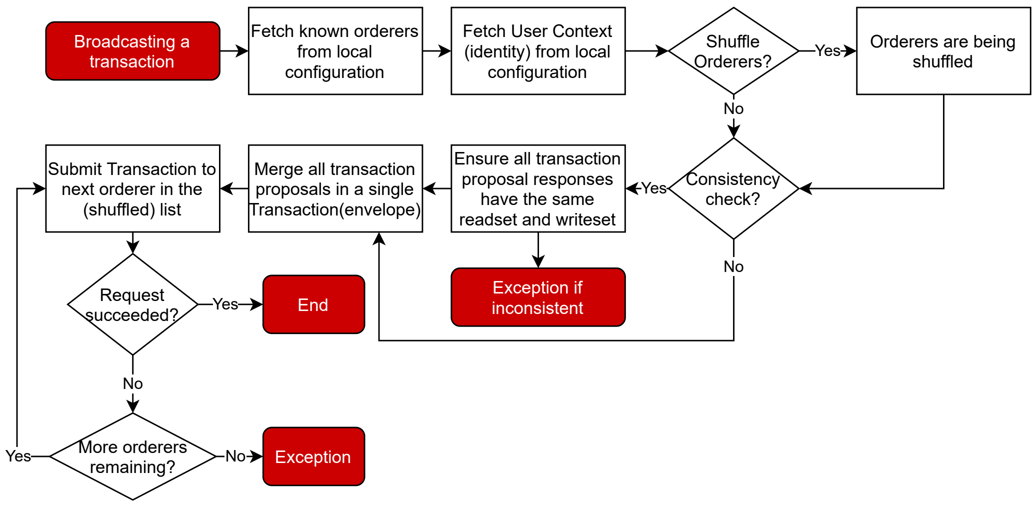

Broadcasting a transaction
Fetch known orderers from local configuration
Fetch User Context (identity) from local configuration
Shuffle Orderers?
Yes
Orderers are being shuffled
No
Consistency check?
Yes
Ensure all transaction proposal responses have the same readset and writeset
Exception if inconsistent
No
Merge all transaction proposals in a single Transaction(envelope)
Submit Transaction to next orderer in the (shuffled) list
Request succeeded?
Yes
End
No
More orderers remaining?
No
Exception
Yes

Fig. 4.5 Broadcasting the Transaction

4.1.5 Phase 5: Processing a Broadcast-request by the Ordering Service

To listen for `broadcast(transaction)`-requests, the ordering service node starts an infinite loop and waits for gRPC-requests to arrive, i.e. a *Handler*. After a `broadcast(transaction)`-request is received by this Handler, the request is forwarded for processing.

The first step is to identify the channel name, used to check that the channel actually exists, and to fetch the channel's blockchain (i.e., only a part of the ledger excluding the world state) for that channel. Consecutively, the channel's configuration is retrieved from this blockchain.

Then, the message itself is being validated by dropping the message if any of the following five conditions are true. (1) First, the message is dropped if the payload is empty, meaning that the transaction can never be executed. (2) Second, the message's size may not exceed a maximum number of bytes configured locally on the orderer. (3) Thirdly, the signature over the transaction is checked to be valid. (4) Then, the client submitting the transaction must be able to satisfy the `Channel.Writers`-policy of the channel it created the transaction for. Typically, this is set to `ANY Writers`, meaning anyone can create transactions and `Broadcast` them. Yet, this could be used to restrict who may send a transaction, and if a channel participant cannot satisfy this policy, the transaction is dropped. (5) Optionally, digital identities may be configured to expire after a certain time of issuance. When enabled, this expiration time is compared with the node's system time. If a digital identity has expired, the transaction is considered invalid and is dropped. These are the only validation steps performed by an OSN. The transaction contents are never inspected.

The transaction is passed to a wrapper for the configured consensus algorithm, further explained in Section 5.1. Instead of forwarding this transaction immediately to the consensus cluster itself, the wrapper is responsible for creating blocks first. Namely, blocks are actually being ordered, rather than transactions. To form blocks, transactions are put in a buffer until the block is *cut*. When the initial transaction eventually returns, the OSN checks if the transaction does not exceed the maximum bytes of the buffer. If so, the old buffer gets cut into a block and a block timer is started to prevent large latencies. Then, our transaction is added to the (empty or non-empty) buffer. The buffer containing our new transaction will be cut into a block once it reaches its maximum size in terms of bytes, its maximum transaction count, or exceeds its maximum delay.

After the block has been cut, it is forwarded to the *actual* consensus cluster, which establishes a total ordering of blocks. The consensus algorithm is omitted in this section, since it is explained in more detail in Section 5.1. At this point in time, the order of blocks is not guaranteed whatsoever. In fact, the block submitted to the consensus cluster may still be overwritten, as is discussed in more detail in Chapter 5.

Eventually, after a block's order has been established, the consensus cluster performs a callback to the OSN. Note that since ordering was not guaranteed while submitting the block to the consensus cluster, it is technically possible a different block (submitted by another OSN) or an older block (of which its order was not committed yet) is returned to the OSN. In any case, all blocks returned by the consensus cluster to the ordering service have been committed and are final. Finally, the blocks are delivered to peers, as is explained in the next phase. Figure 4.6 shows the steps an OSN takes during this phase.

Note that during the entirety of this phase, the ordering service does not inspect nor validate the semantics of the received transaction. Therefore, an invalid transaction may be ordered into blocks. Even though the OSNs do not inspect transaction contents, they *do* receive this information and would be able to intercept/parse this. As we will see later, private data is *not* seen by the ordering service, highlighting why isolated application channels may not be a sufficient strategy for achieving privacy.

4.1.6 Phase 6: Block Delivery and Gossip Protocol

The previous steps discussed how a transaction proposal is created, endorsed, and submitted to the ordering service. This ordering service orders all transactions and uses these transactions to create and order blocks using a consensus protocol

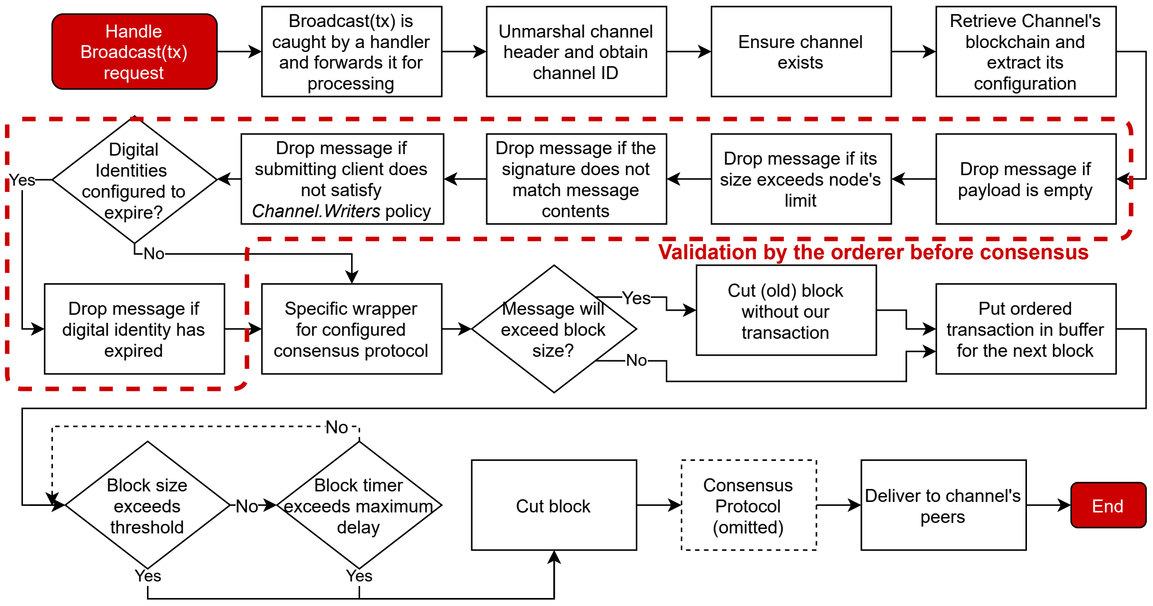

Handle Broadcast(tx) request
Broadcast(tx) is caught by a handler and forwards it for processing
Unmarshal channel header and obtain channel ID
Ensure channel exists
Retrieve Channel's blockchain and extract its configuration
Drop message if payload is empty
Drop message if its size exceeds node's limit
Drop message if the signature does not match message contents
Drop message if submitting client does not satisfy *Channel.Writers* policy
Digital Identities configured to expire?
Yes
No
Validation by the orderer before consensus
Drop message if digital identity has expired
Specific wrapper for configured consensus protocol
Message will exceed block size?
Yes
No
Cut (old) block without our transaction
Put ordered transaction in buffer for the next block
Block size exceeds threshold
No
Block timer exceeds maximum delay
No
Yes
Yes
Cut block
Consensus Protocol (omitted)
Deliver to channel's peers
End

Fig. 4.6 Processing a broadcast request by the Ordering Service

(Section 5.1). Now, we zoom back into the peer nodes to investigate how they receive the latest blocks to update their ledger.

Gossip Protocol

During the start-up of a peer, every peer initializes a *Gossip Service*. This is a service used for retrieving and disseminating blocks from the ordering service within its own organization. Every peer has exactly one gossip service, which coordinates the block delivery of all the channels that the peer is connected to. Within an organization and for each channel, there is *at least one* leader. This leader is a peer that frequently polls the ordering service to retrieve new blocks, and shares these blocks to non-leaders within the peer's organization. Since different peers may be connected to different channels, a peer may be a leader for one channel, while not being a leader for another channel.

There are two slight variants of the Gossip protocol: an organization may statically define one or multiple leaders, or leaders may be dynamically elected during runtime. By statically defining leaders, it is technically allowed that an organization with 2 or more peers consists only of leaders (and zero non-leaders). This results in all peers individually fetching blocks from the orderer. Although perfectly valid, this increases the (network) load of the ordering service linearly with the amount of peers in each organization. Instead, the amount of leader peers within an organization should be kept to a minimum, distributing the blocks mostly peer-to-peer, to minimize the orderer's load. However, a risk of defining only few leaders is that if they become unavailable, the entire organization not receiving blocks.

To combine the best of both worlds, i.e. few peers per organization without the risk of all peers being down, the gossip protocol of Fabric supports leader election. This means that all electable leaders regularly send heartbeats to each other. Then, the electable peers reach agreement by deciding on a leader with active heartbeats. Normally, an organization with leadership election always has exactly one leader. Yet, when a network partition occurs and not all peers can reach each other, a new leader is elected for each partition. Thus, this always ensures all peers receive the latest blocks from the ordering service.

Once a leader has been elected, the gossip protocol starts the *Delivery Service* for the particular channel the peer is a leader of. The peer starts by fetching the blockchain's height, i.e. the last block number it knows of. This block number is included in a `SeekInfo`-request. The peer submits this request to the ordering service, to tell it the last block it knows of (and thus, also the information it lacks). To send this request to the ordering service, the leader peer randomly connects to one of the OSNs, as defined in the channel's configuration. The `SeekInfo`-request with the peer's last-known block is sent to the orderer only once, since the connection remains open. Every time the orderer has finished creating a block, the new block is submitted to (all) the leader(s) over all its open connections. This resembles a publish/subscribe pattern.

Processing blocks received through delivery

Once a block has been received by a leader, the leader performs various verification steps. First, the block number defined in the message's header must match the block number inside the block itself. Next, the exact channel defined inside the block must match the channel of the peer's delivery service, to prevent adding a block meant for a different channel. Furthermore, the block's hash is recomputed and compared with the hash of the message to ensure that the block's contents were not altered in transit. Finally, the `Channel/Orderer/BlockValidation`-policy is evaluated. Essentially, this policy specifies what orderer must have signed in order to be considered valid. By default, this is set to `ANY Writers`, meaning a block is considered valid as long as it is signed by any of the orderers defined in the channel's configuration.

After a block has been verified by the leader, the leader will perform two actions. The first action is to push the block to a so-called `Payloads`-buffer, which is a queue of blocks waiting to be appended to the ledger. Secondly, the leader performs the actual gossip of the block. This gossiping is implemented using batches: the block is again stored in a buffer. Once the buffer reached a threshold of amount of blocks, or the first block in the buffer has been added in the buffer for a configurable amount of time, the gossip-buffer is emitted to `propagatePeerNum` nodes. This decision making in terms of number of blocks and timeout is very reminiscent of what we have seen for cutting blocks. Figure 4.7 illustrates and summarizes this phase.

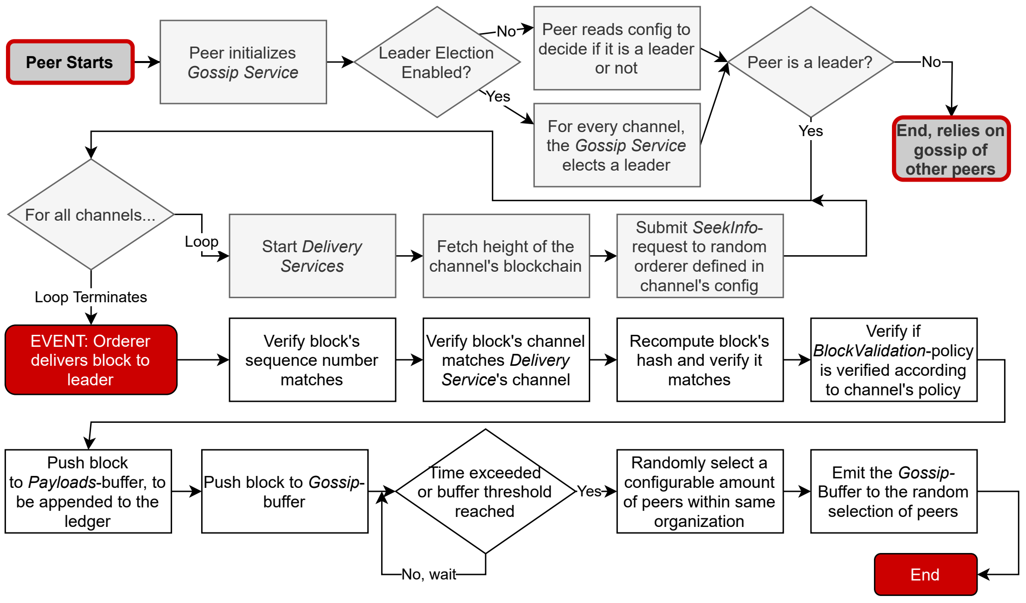

Peer Starts
Peer initializes Gossip Service
Leader Election Enabled?
No
Yes
Peer reads config to decide if it is a leader or not
For every channel, the Gossip Service elects a leader
Peer is a leader?
No
Yes
End, relies on gossip of other peers
For all channels...
Loop
Loop Terminates
Start Delivery Services
Fetch height of the channel's blockchain
Submit SeekInfo-request to random orderer defined in channel's config
EVENT: Orderer delivers block to leader
Verify block's sequence number matches
Verify block's channel matches Delivery Service's channel
Recompute block's hash and verify it matches
Verify if BlockValidation-policy is verified according to channel's policy
Push block to Payloads-buffer, to be appended to the ledger
Push block to Gossip-buffer
Time exceeded or buffer threshold reached
Yes
No, wait
Randomly select a configurable amount of peers within same organization
Emit the Gossip-Buffer to the random selection of peers
End

Fig. 4.7 Block Delivery from the Ordering Service to Peers

The observant reader may have noticed that a leader peer only emits the block to a *maximum* (`propagatePeerNum`) number of peers, rather than *all* peers in its organization. This is a deliberate design choice to prevent a single entity from continuously having to perform expensive broadcasts. This distribution of blocks resembles the *push* communication pattern. Also worth mentioning is that peers occasionally try to *pull* blocks from random other peers. Although this results in *eventual consistency* in terms of available blocks, a gossiping protocol involving both push and pulls is known to be a very scalable approach for data dissemination in distributed systems [37]. However, a small disclaimer applies: since the gossip protocol relies on stochastic techniques to disseminate blocks to (pseudo-)randomly selected peers, there exists a (small) chance that a certain subset of peers are skipped during the push phases, while trying to pull blocks from nodes in the same subset without the new block. As such, block delivery is mathematically not guaranteed, but as time approaches infinity, the probability of receiving a block becomes 1. The fact that Fabric uses a combination of push and pull operations also help speed up the block propagation in practice. Figure 4.8 illustrates the Gossip Protocol.

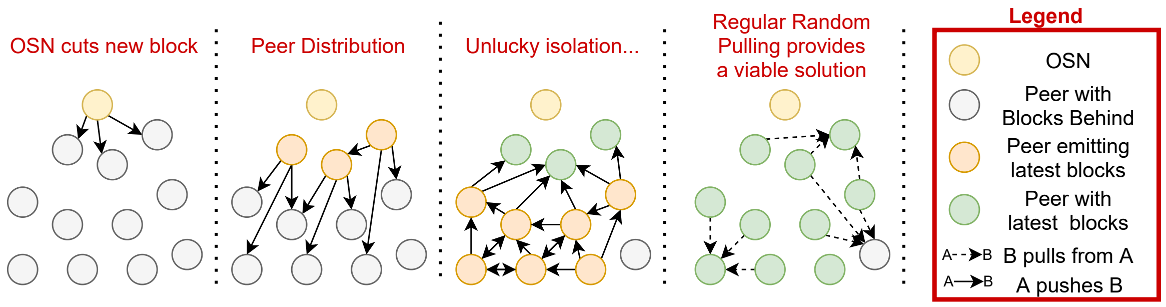


Fig. 4.8 Example of Gossip Protocol

4.1.7 Phase 7: Peer Block Validation

For every channel a peer has joined, it creates an infinite loop to continuously check if new blocks have entered the `Payloads`-buffer, as mentioned previously in Section 4.1.6. This `Payloads`-buffer has a queue-like structure, in which blocks with the highest sequence numbers are added last and blocks with the lowest sequence number are at the start of the queue. Thus, this loop first fetches a block from the queue with lowest sequence number and unmarshals it.

Then, multiple validation steps are performed on the block by the peer. The peer verifies that the block contains both a header and data-field (containing the transactions). Next, all transactions within a single block are being validated in parallel. This starts by unmarshalling each transaction and verifying that its structure is correct. Then, each transaction creator's identity is fetched and validated using the channel's MSP. After it has been established that the creator's identity is valid, it is used to check if the transaction's cryptographic signature is valid.

Recall that a transaction contains a set of endorsements, as illustrated in Figure 3.8 and as explained in Section 4.1.3 as part of the transaction flow. Thus, the peer sequentially validates that the signature of each proposal is valid. Now, the transaction is considered structurally valid and some more validations are performed on the semantic level: the channel mentioned in the transaction must be existent and matching the block, and the transaction may not be a duplicate that the peer already processed. Lastly, a plugin is called to evaluate whether the transaction satisfies the endorsement policy (including all chaincodes on the same channel it invokes as well). This plugin is also referred to as the Validation System Chaincode (VSCC). After all of these steps, every transaction is marked as valid or invalid. That marks the end of the parallel validation of all transactions in a block, as well as the block itself. The flow of this phase is captured by Figure 4.9.

4.1.8 Phase 8: Commit

Finally, after a block and its transactions have been validated, we have arrived at the block commit phase. As a first step, the block is constructed by only including transactions previously marked as valid. For every (so-far) validated transaction, a read-write conflict check is performed. This is one final check on the transaction, also known as Multiversion Concurrency Control (MVCC). At this stage, a transaction can still be marked as invalid if at least one of the reads was performed on a different version of a key in the key-value store. This step ensures that a transaction may not modify the ledger if the writeset was generated using an outdated ledger state. This prevents the well-known double-spend problem. Moreover, this check could not have been performed in the previous phase, as all transactions in a block are validated concurrently: Two consecutive transactions in a block could read and modify the same key, requiring sequential checks. After passing the MVCC check, the transaction's writes are added to an in-memory set with key updates. The new version for these writes is

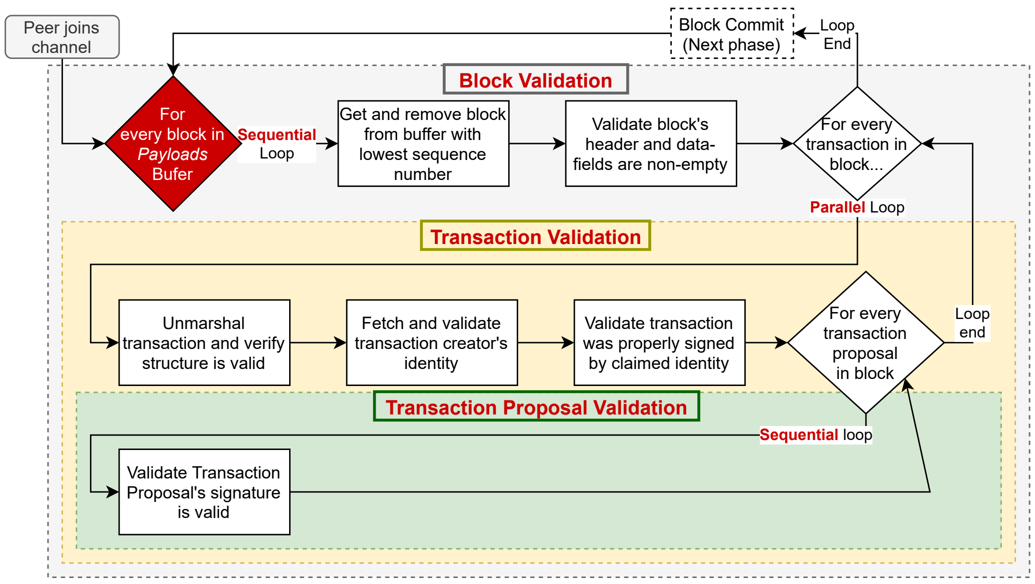

Peer joins channel
Block Commit (Next phase)
Loop End
Block Validation
For every block in Payloads Bufer
Sequential Loop
Get and remove block from buffer with lowest sequence number
Validate block's header and data-fields are non-empty
For every transaction in block...
Parallel Loop
Transaction Validation
Unmarshal transaction and verify structure is valid
Fetch and validate transaction creator's identity
Validate transaction was properly signed by claimed identity
For every transaction proposal in block
Loop end
Transaction Proposal Validation
Sequential loop
Validate Transaction Proposal's signature is valid

Fig. 4.9 Block Validation flow

a combination of the block height, together with the transaction's index within the block. These writes are not immediately applied to the ledger, since these changes should be performed atomically once per block.

To ensure that the values provided in the in-memory updates are compatible with the various world state storage implementations (LevelDB / CouchDB), all updates are converted to bytes. To perform the actual updates, the ledger's blockchain is locked first. Then, the new block is only appended to the ledger if there was no issue with the previous block. Otherwise, something may have gone wrong earlier (such as a crash during the writing of the previous block). If the blockchain on the peer's file system is technically a fork (i.e. corrupted), this is easily solved by dropping the invalid blocks and retrieving new blocks again from the ordering service or other peers.

After the new block has been written to the file system's blockchain as an append-only file, the `stateDB` is to be updated. Similarly to what we have seen previously, the entire key-value store is being locked. Luckily, both state database implementations, LevelDB and CouchDB, support atomic batch updates. The peer submits the updates to the key-value store as a batch using atomic operations supported by the key-value storage engine. This ensures that either all updates are written to the state database, or none are. After the world state has been updated, the ledger's blockchain and state locks are released. This phase is captured by Figure 4.10, but also denotes the end of Fabric's transaction flow.

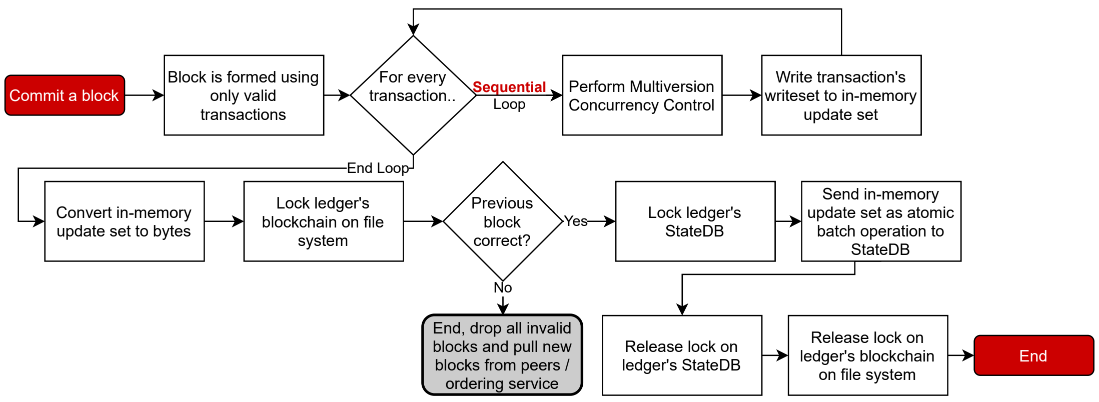

Commit a block
Block is formed using only valid transactions
For every transaction..
Sequential Loop
Perform Multiversion Concurrency Control
Write transaction's writeset to in-memory update set
End Loop
Convert in-memory update set to bytes
Lock ledger's blockchain on file system
Previous block correct?
Yes
No
Lock ledger's StateDB
Send in-memory update set as atomic batch operation to StateDB
End, drop all invalid blocks and pull new blocks from peers / ordering service
Release lock on ledger's StateDB
Release lock on ledger's blockchain on file system
End

Fig. 4.10 Committing a block to the ledger

4.2 Transactions in Besu

This section follows a similar approach to Section 4.1: the public transaction flow is followed from the moment a transaction is created to when it is appended to the ledger. Recall that as discussed in Section 3.2.3, there exist two different types of transactions, namely *Message Calls* that transfer a cryptocurrency or invoke a smart contract, and *Contract Creation*-transactions. In that same section, we already saw that both types of transactions are recorded on the ledger using the same data structure. As their transaction flows are rather similar, the focus of this section is on the invocation of a smart contract through Message Call-transactions. Additionally, the transaction flow assumes the JSON-RPC endpoints are used, since this message protocol is also supported by most other Ethereum Clients[4], allowing a future researcher to directly compare Besu's transaction flow with another Ethereum Client's flow.

As explained in Section 3.2.6, there exist many approaches to creating transactions using various tools and libraries. Yet, often the JSON-RPC endpoint is (indirectly) called regardless of which tool or library is used. The remainder of this section assumes `Web3.js`[5] is used with Solidity. Furthermore, it is worth mentioning that Gas-related activities are also included in this flow. Even though Gas is often set to be free, a permissioned Besu network might still enable fees if they wish to do so.

On a high level, Besu's transaction flow looks as following. First, an actor locally creates a transaction, as elaborated on in Section 4.12. But since a Besu node only accepts signed transactions, an actor must sign this locally or outsource this signing to an EthSigner instance (Section 4.13). When a Besu node then receives a signed transaction, the node validates the transaction and performs various forms of Access Control (Section 4.2.3). This transaction is then added to a Transaction Pool that is local to the node, after which the transaction is broadcasted to other parties (Section 4.2.4). In another thread, the consensus protocol decides when a node should create a block. When invoked by the consensus protocol, a node creates a block by only including transactions that were simulated (executed) successfully. This new block is then returned back to the consensus protocol for reaching consensus and dissemination, as described by Section 4.16. Finally, after receiving a block, the ledger's world state and blockchain is updated (Section 4.2.6). This high-level flow is also visualized in Figure 4.11.

4.2.1 Phase 1: Transaction Creation

Before a transaction can be created, the actor needs to define the contract's interface locally using an Application Binary Interface (ABI)-format. This ABI is essentially a JSON-structure containing a list of (invocable) functions, their parameters, outputs, name and more. An example of a very trivial ABI that defines a single function with as only argument a number is provided in Listing 4.3. This interface is used to inform the actor of the functions it can invoke, and the parameters required by each function. At the time of writing, however, Besu does not expose an endpoint to retrieve the ABI at runtime, and neither does the more popular Go Ethereum implementation. Therefore, an actor must provide this ABI by him or herself. The only ways to retrieve the ABI are (1) as output of compiling the smart contract's source code, (2) by fetching it from a block explorer, or (3) by manually writing it[6]. Even when using an SDK, one needs to specify the contract's ABI when making a transaction: Only mentioning the function to execute is not enough. As is implied by the name of the ABI, it is an interface and not an actual implementation or reference of a contract on the ledger.

```
myABI = [
  {
    "inputs": [
      {
        "internalType": "uint256",
        "name": "val",
        "type": "uint256"
      }
```

[4]Other Ethereum clients supported JSON-RPC are CPP-Ethereum, Go-Etheruem, Py-Ethereum, Parity as shown on https://eth.wiki/json-rpc/API.
[5]https://github.com/ChainSafe/web3.js
[6]Manually writing it can be tedious and very prone to errors, as the contract gets larger, yet nothing prevents one from doing so.

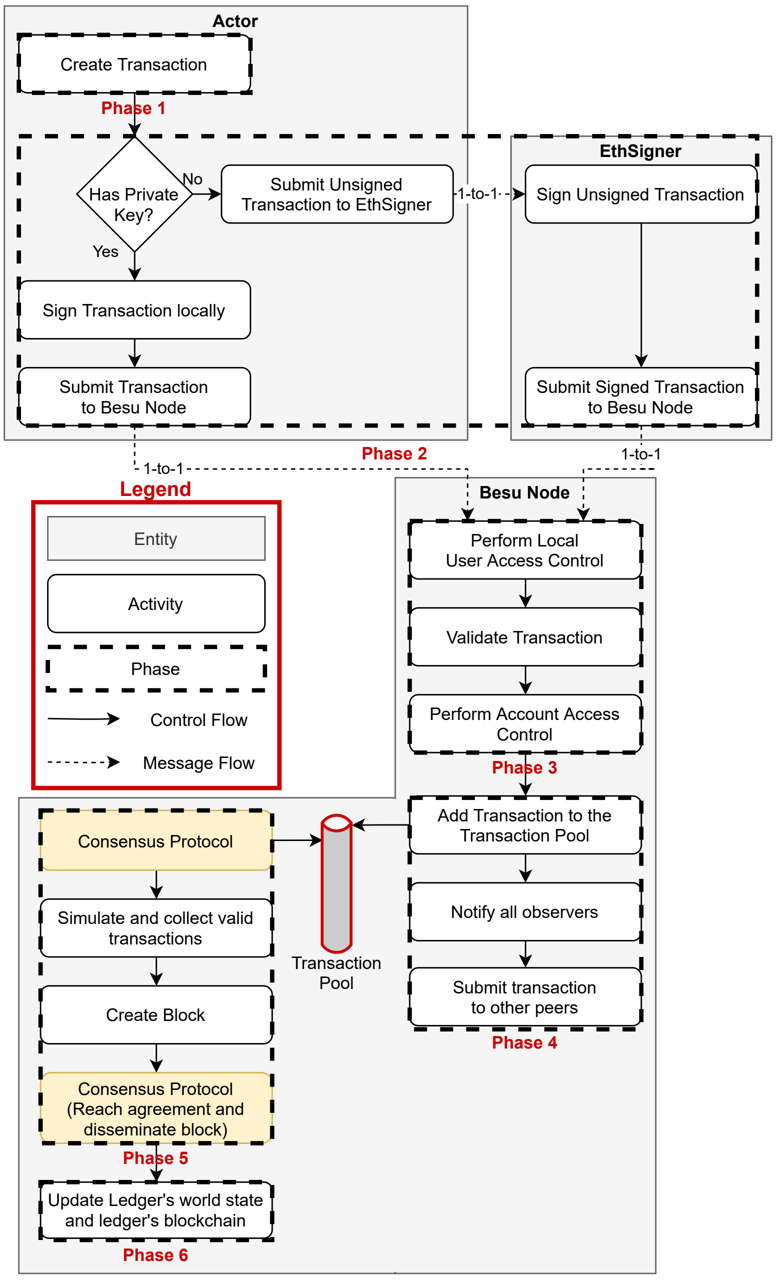

Actor
Create Transaction
Phase 1
Has Private Key?
No
Yes
Submit Unsigned Transaction to EthSigner
1-to-1
EthSigner
Sign Unsigned Transaction
Sign Transaction locally
Submit Transaction to Besu Node
Submit Signed Transaction to Besu Node
Phase 2
1-to-1
1-to-1
Legend
Entity
Activity
Phase
Control Flow
Message Flow
Besu Node
Perform Local User Access Control
Validate Transaction
Perform Account Access Control
Phase 3
Add Transaction to the Transaction Pool
Notify all observers
Submit transaction to other peers
Phase 4
Consensus Protocol
Transaction Pool
Simulate and collect valid transactions
Create Block
Consensus Protocol (Reach agreement and disseminate block)
Phase 5
Update Ledger's world state and ledger's blockchain
Phase 6

Fig. 4.11 Hyperledger Besu's transaction flow

```
    ],
    "name": "setNumber",
    "outputs": [],
    "stateMutability": "nonpayable",
    "type": "function"
  }
];
```

Listing 4.3 Example of an Application Binary Interface

After a local contract instance has been created by passing the ABI to Web3.js, the actor can interact with it as if it invokes the methods directly, as can be seen in Listing 4.4. The result of this local invocation is called *Transaction Data*. Up to this point, the actor has not connected to the Besu network at all (such as to a node / ethsigner). Thus, the transaction data is simply a *request* encoded in hexadecimal to invoke a deployed contract's function.

```
myContract = new web3.eth.Contract(myABI);
txData = myContract.methods.setNumber(5).encodeABI();
```

Listing 4.4 Web3.js contract invocation

To create the actual transaction with other relevant fields, the transaction creator defines a so-called *Transaction Object*. This is an object that specifies all details required by a Besu client (node) in order to properly execute a transaction. An example of such a transaction object is shown in Listing 4.5. In this transaction object, we see that the public address of the EOA that creates the transaction must be defined (`from`), as well as the smart contract to invoke (`to`). A `nonce` is optionally specified. Essentially, this nonce defines the n-th transaction created by an account and is used for establishing an account-based ordering to prevent double-spends. In most cases, this nonce can be automatically derived to relieve the actor from having to keep track of this value. Explicitly specifying it may be used to override a transaction that is pending at a node and not yet included in a block.

Then, a `gasPrice` is passed. This is a weight to determine the amount of cryptocurrency that should be deducted per unit of gas used. Since the need for a cryptocurrency as financial incentive is often absent for permissioned networks, it is common to set this value to 0. This means that nodes never (try to) charge a fee for including the transaction in a block. The `gas`-field is used to specify how much gas the execution of a transaction may consume at a maximum, before the transaction is aborted. This notion of `gas` is entirely decoupled from a cryptocurrency: *Gas* is the amount of fuel (operations) required for executing transactions, whereas the underlying *Crypto* is used to pay for that fuel. A `value`-field is used to define an amount of cryptocurrency to transfer from the EOA's wallet to another account's wallet (in this case, a SCA). Similar to the gas price, this value is often set to 0 in a permissioned setting. Lastly, a `data`-argument should be provided by the actor. In fact, this is the *Transaction Data* mentioned earlier, i.e. a hexadecimal string that encodes the function name to call and its arguments. This phase is illustrated by Figure 4.12.

```
txObject = {
    "from":"0xfe3b557e8fb62b89f4916b721be55ceb828dbd73",
    "to":"0xd46e8dd67c5d32be8058bb8eb970870f07244567",
    "nonce":"0x0", // Optional, default is inferred from account
    "gasPrice":"0x0", // Optional, default is 0
    "gas": "0x7600", // Optional, default is 0x15F90 (90.000)
    "value": "0x0", // Optional, default is 0
    "data":"0xf869018203e88..."
}
```

Listing 4.5 Transaction Object for Web3js-eea

4.2.2 Phase 2: Transaction Signing and Submission

Now, the actor is given two choices depending on its knowledge and the network composition: either the transaction is signed locally, or the transaction is signed by an EthSigner instance. Figure 4.13 summarizes both flows. Whether a transaction was signed locally or by an EthSigner instance, a signed transaction is submitted to a Besu node at the end. Since both flows result in the same type of request to a Besu node, the flows are merged back to a single flow again at the end of this phase.

Signing the transaction locally

In the simplest case, an actor has access to the account's private key. A first step of the actor is to provide his private key in the Web3.js instance's wallet. Additionally, the actor needs to configure Web3.js to connect to a Besu node by providing its address and port. Before sending the data, the local Transaction Object is signed using the actor's private key. To

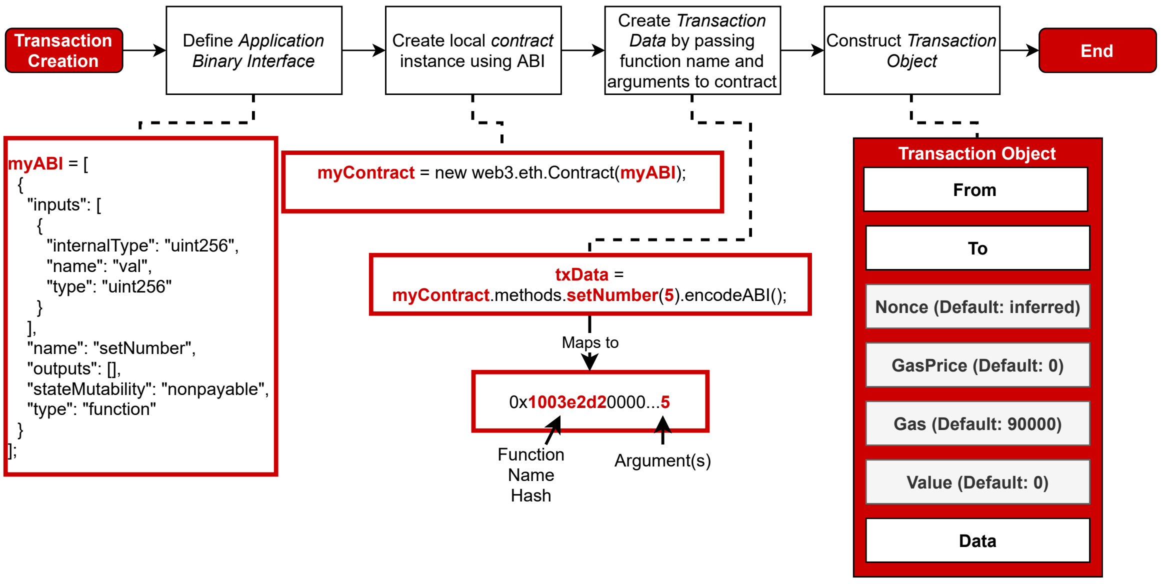

Transaction Creation
Define Application Binary Interface
Create local contract instance using ABI
Create Transaction Data by passing function name and arguments to contract
Construct Transaction Object
End
myABI = [
 {
  "inputs": [
   {
    "internalType": "uint256",
    "name": "val",
    "type": "uint256"
   }
  ],
  "name": "setNumber",
  "outputs": [],
  "stateMutability": "nonpayable",
  "type": "function"
 }
];
myContract = new web3.eth.Contract(myABI);
txData = myContract.methods.setNumber(5).encodeABI();
Maps to
0x1003e2d20000...5
Function Name Hash
Argument(s)
Transaction Object
From
To
Nonce (Default: inferred)
GasPrice (Default: 0)
Gas (Default: 90000)
Value (Default: 0)
Data

Fig. 4.12 Creating a transaction for Besu

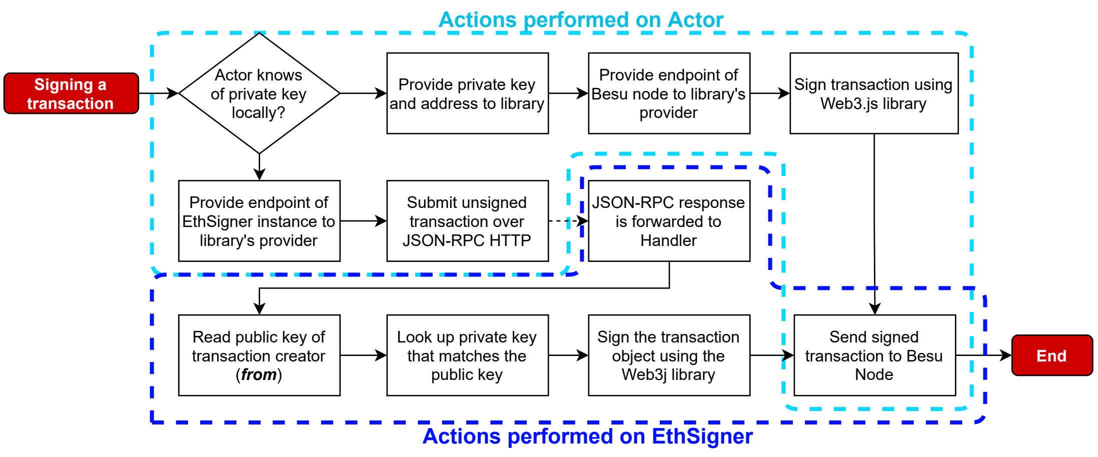

Actions performed on Actor
Signing a transaction
Actor knows of private key locally?
Provide private key and address to library
Provide endpoint of Besu node to library's provider
Sign transaction using Web3.js library
Provide endpoint of EthSigner instance to library's provider
Submit unsigned transaction over JSON-RPC HTTP
JSON-RPC response is forwarded to Handler
Read public key of transaction creator (from)
Look up private key that matches the public key
Sign the transaction object using the Web3j library
Send signed transaction to Besu Node
End
Actions performed on EthSigner

Fig. 4.13 Signing a transaction object for Besu

actually submit the signed transaction to the node, `web3.eth.sendSignedTransaction()` is invoked which submits the transaction to the node. Listing 4.6 demonstrates how a self-signed transaction in Besu is submitted directly to a node. Although the Web3.js library only supports specifying a single node's endpoint at a time, one could change this endpoint on the fly and connect to another endpoint in case a node is unavailable.

```
web3.eth.accounts.wallet.add({ // Configure wallet with private key
    privateKey: '0x1234...',
    address: '0x5678...'
});

// Configure endpoint to connect to
web3.setProvider('http://localhost:8545');

web3.eth.signTransaction(txObject, myPublicAddress).then((txDataSigned) => {
  web3.eth.sendSignedTransaction(txDataSigned);
});
```

Listing 4.6 Submitting a Signed transaction

Signing the transaction By EthSigner

An alternative to self-signing transactions, is to forward an unsigned transaction to an EthSigner instance in the network. To do so, the actor only has to perform two very trivial steps. The first is to simply configure the library to connect to the EthSigner node (as if it was an actual Besu node). Then, after the library knows how to connect to a node, the actor can invoke the `sendTransaction()`-function. This sends an HTTP-request to the EthSigner instance. A small sample of code is given in Listing 4.7.

```
web3.setProvider('http://localhost:8545');
web3.eth.sendTransaction(txObject);
```

Listing 4.7 Submitting an Unsigned Transaction

Every JSON-RPC request follows the same RPC-structure, regardless of the method it invokes, as was explained in Section 3.2.6 and shown in Listing 3.5. Once the EthSigner receives a request, it inspects the `method` and forwards it to the corresponding (`SendTransaction`)Handler. This handler inspects the public key of the EOA in the transaction object and looks up its private key. This private key is used to sign the transaction using Web3j. Consecutively, it is parsed to a JSON-RPC-request which is sent to a Besu node. The address and port of this node was provided earlier when the EthSigner-instance was started. Each EthSigner instance only defines the (signed) endpoint of a single Besu node. Thus, it means that if only the Besu nodes becomes unavailable, the EthSigner instance linking to it also non-functional. This introduces a single point of failure. To overcome this, the actor may require multiple EthSigner instances per Besu node. If a connection then fails, the actor can simply change the provider's endpoint and connect to a different EthSigner instance. (In Section 6.2, we see this requirement is even stronger, i.e. every single node in a private group must be online)

4.2.3 Phase 3: Receiving a Signed Transaction Request

Every single JSON-RPC-request is first processed by checking if authentication is enabled on the node. If this is the case, a JWT is read from the `Authorization` request header. Recall from Section 3.2.7 that these are generated by logging in with a username and password[7]. The JWT is decoded in a username-password combination. Since JWTs tokens are only valid temporarily, the node checks if it has not expired yet. Finally, the requested method (in this case, `eth_sendRawTransaction`) is extracted from the JSON-RPC request, and is used together with the username and password to see if the user is authorized to perform the requested operation.

Similarly to how the EthSigner instance received a JSON-RPC request, a Besu node routes all requests to their corresponding handler based on their method name. However, since a Besu node now receives a *signed* transaction as opposed to an unsigned one, the node now processes an `eth.SendRawTransaction()`-request. An example of such a request was given in Listing 3.7.

The transaction object is received by the node in a nontrivial way, as its *structure*[8] was encoded using RLP [23]. This encoding technique is required since the transaction data may be of variable length, i.e. the method invocation could have the length of 10 bytes or 200. Without explicitly encoding structure, the node does not know where a field of variable length starts and ends. After the encoded bytes have been decoded according to their structure, the application can now start parsing and reading individual data types (such as an integer or a list). First, the format of the transaction is determined. In fact, the transaction structure may differ for different Ethereum protocol releases, as denoted by the EIP. After the node has established what the (presumed) structure of the transaction is, the bytes are parsed to a Java Transaction instance.

Next, various validation steps are performed on the transaction object. First, the transaction's signature is validated. Then, the node computes an estimate of the amount of Gas required (based on the number of bytes in the `payload`), and compares

[7]The login process returns a token that acts as a session. Since logging in is not required every time a transaction is submitted, this process is not included in the transaction flow.

[8]Recursive Length Prefix (RLP) encodes structure, instead of individual data types (such as String or integer). Examples can be found at: https://eth.wiki/fundamentals/rlp#definition

this with the transaction's Gas limit. Naturally, if the Gas limit is less than the amount of gas required, the transaction is marked as invalid. Furthermore, the transaction's gas limit must be lower than the block's gas limit as defined in the Genesis file. Finally, the transaction is being validated in relation to the account's address: the account must have a sufficient amount of Gas, and the transaction's nonce must be at least the EOA's nonce. If account-based permissioning is enabled (Section 3.2.7), the node validates whether the account is allowed to create a transaction by consulting the local or network whitelist of accounts. This marks the end of the transaction validation. Only if the transaction is valid after all these checks, control flow continues to the next phase. This phase is illustrated by Figure 4.14.

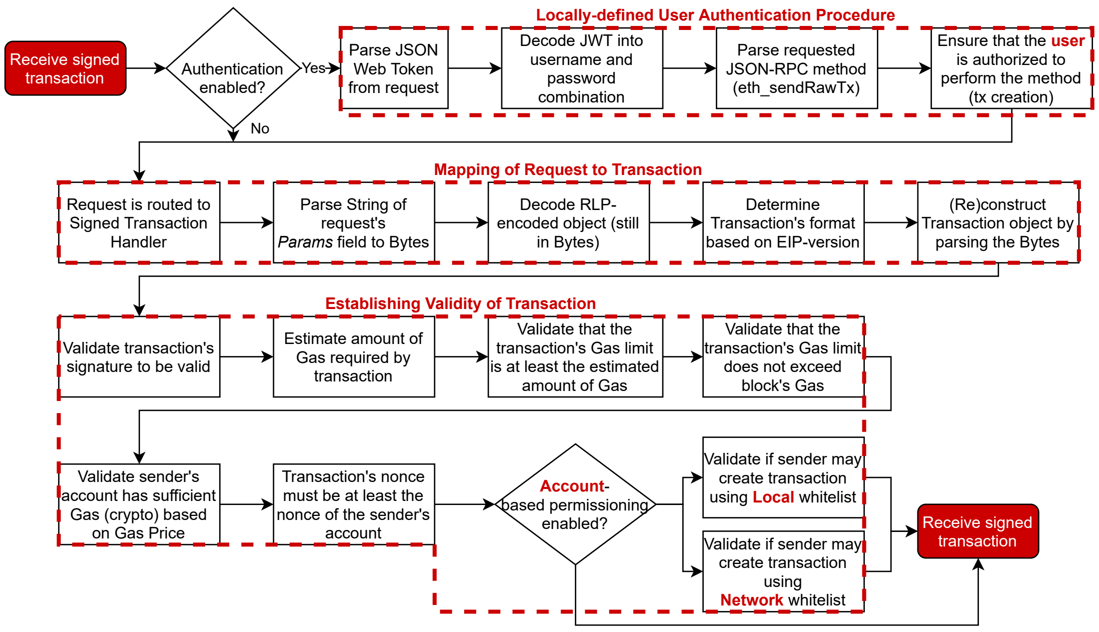

Receive signed transaction
Authentication enabled?
Yes
No
Locally-defined User Authentication Procedure
Parse JSON Web Token from request
Decode JWT into username and password combination
Parse requested JSON-RPC method (eth_sendRawTx)
Ensure that the user is authorized to perform the method (tx creation)
Mapping of Request to Transaction
Request is routed to Signed Transaction Handler
Parse String of request's Params field to Bytes
Decode RLP-encoded object (still in Bytes)
Determine Transaction's format based on EIP-version
(Re)construct Transaction object by parsing the Bytes
Establishing Validity of Transaction
Validate transaction's signature to be valid
Estimate amount of Gas required by transaction
Validate that the transaction's Gas limit is at least the estimated amount of Gas
Validate that the transaction's Gas limit does not exceed block's Gas
Validate sender's account has sufficient Gas (crypto) based on Gas Price
Transaction's nonce must be at least the nonce of the sender's account
Account-based permissioning enabled?
Validate if sender may create transaction using Local whitelist
Validate if sender may create transaction using Network whitelist
Receive signed transaction

Fig. 4.14 Receiving a Signed Transaction Request on a Besu node

4.2.4 Phase 4: Adding to Transaction Pool

Blocks in Besu are (eventually) created by individual nodes. What transactions are included in a block and their ordering is thus decided on an individual basis. To facilitate this process, each node maintains its own *Transaction Pool*, which takes the form of a priority queue. As a first step, control flow enters a critical section (using `synchronized`), meaning that a lock is obtained on the priority queue. If the transaction is already waiting in the queue, it is a duplicate and the transaction is discarded. Then, if there is already another transaction from the same sender with the same nonce, that transaction is discarded as it should be overridden.

After these steps, the new transaction is now added to the priority queue. A transaction's priority is established based on multiple fields. Transactions directly submitted to this node take highest precedence. Then, transactions with the highest Gas price are preferred, as a node (in a permissionless network) is greedy and wishes to receive the largest fee. Lastly, if two transactions are submitted in the same manner (locally / remote) and their fee is the same, priority is given to transaction the node received first. If the maximum amount of pending transactions is exceeded, transactions with lowest priority are discarded. Finally, the lock on the priority queue is released.

After this transaction has been added to the transaction pool, the Observer design pattern is used to notify all observers of this new transaction over WebSockets. Similarly and after this step, all observers are notified of all dropped transactions (if any) due to the priority queue being full. This could be used, for instance, if one is waiting for their transaction to be included in a block. Then, if the actor is notified that its transaction is dropped (at the current node!), it may decide to increment its Gas fee in a permissionless network. Lastly, a job is scheduled to submit the new transaction to all other peers that the node has established an active connection with. This transaction is submitted to other peers using the Ethereum Wire Protocol[9], which is built on top of the RLPx transport protocol[10] based on Transmission Control Protocol (TCP). After this step, a JSON-RPC response is returned to the actor submitting the transaction to the node. Figure 4.15 visualizes this phase.

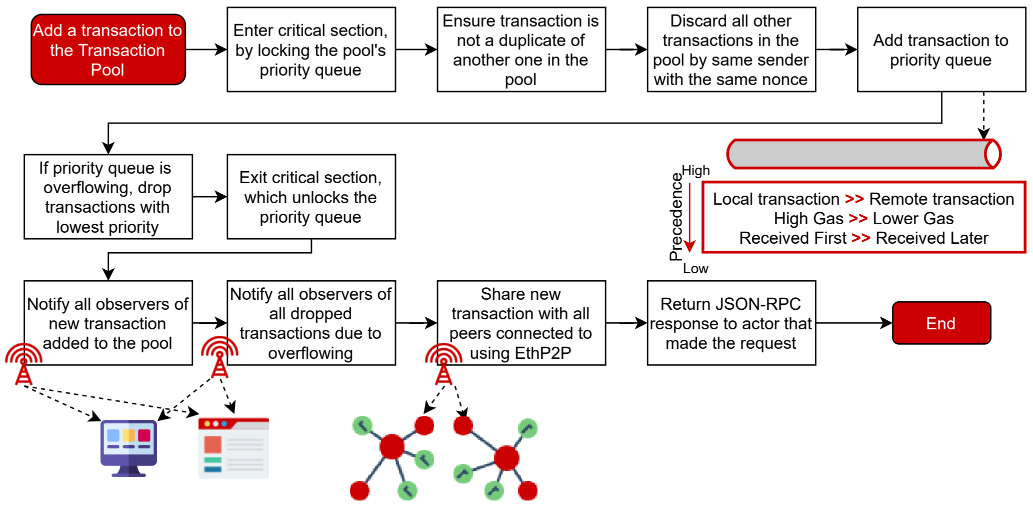

Add a transaction to the Transaction Pool
Enter critical section, by locking the pool's priority queue
Ensure transaction is not a duplicate of another one in the pool
Discard all other transactions in the pool by same sender with the same nonce
Add transaction to priority queue
If priority queue is overflowing, drop transactions with lowest priority
Exit critical section, which unlocks the priority queue
High
Precedence
Low
Local transaction >> Remote transaction
High Gas >> Lower Gas
Received First >> Received Later
Notify all observers of new transaction added to the pool
Notify all observers of all dropped transactions due to overflowing
Share new transaction with all peers connected to using EthP2P
Return JSON-RPC response to actor that made the request
End

Fig. 4.15 Adding a transaction to a node's transaction pool in Besu

4.2.5 Phase 5: Transaction Execution and Block Creation

Every node runs one of the four consensus protocols (Proof-of-Work Ethash, Clique, IBFT 1.0, IBFT 2.0) as described in Section 5.2. These processes run concurrently to the flow described in the previous phases so far[11]. The block creation process is actually called by the consensus protocol, regardless of the consensus implementation used. Moreover, the block creation process itself differs slightly for each consensus protocol. Yet, on an abstracted and higher level, the block creation process is similar. Thus, this phase will not focus on the *exact* details of how a block is created, but is slightly generalized.

As a first step, the Block's header is constructed. This header consists of a hash of the blockchain's head to form a linked list. The new block's gas limit is computed by safely adjusting the head block's gas limit slightly towards the *target gas limit*, provided as commandline argument. In fact, the block's gas limit is allowed to change a small (maximum) amount for every new block that is produced. Thus, if many participants define a high gas limit, and only a few participants define a low gas limit per block, this will result in the block's gas limit increasing over time. This is exactly what can be observed on the MainNet[12]. Note that this in essence is related to a maximum of operations per block, and thus, the amount of transactions. Additionally, a block difficulty is added from the Genesis file (usually minimally configured for permissioned networks). A `coinbase` denotes the address to pay transaction fees to (again, usually ignored for permissioned networks). Lastly, the block number is increased by one and a timestamp is added.

After the block header is constructed, various pieces of information are fetched. First, the world state is updated to match the head of the blockchain. This is performed by a combination of rollbacks (when another chain became longer and is now the new head, the state is reverted to a prior block) and rollforwards (to update the world state to the a newer block). Every rollforward/rollback consists of three operations: updating accounts, updating code, and updating storage entries. For every update operation, an address, old value, and new value must be provided. One can see how storing both the old and new value at every block makes rollbacks and rollforwards easy, as transactions do not have to be reverted/executed

[9]https://github.com/ethereum/devp2p/blob/master/caps/eth.md
[10]https://github.com/ethereum/devp2p/blob/master/rlpx.md
[11]They are always running, regardless of whether transactions are running.
[12]https://etherscan.io/chart/gaslimit

again. After the world state is updated to the new head of the blockchain (before a new block is created), a reference to this world state is returned. This world state, however, is a disposable copy and is only used to create the block.

Next, transactions are selected from the priority queue of the transaction pool. In order to do so, the priority queue is locked again using critical sections. Then, transactions are added from highest priority to lowest priority until there are no more transactions left in the pool or the block's gas limit is reached. After as many transactions are added as possible to the block, all selected transactions are sequentially executed on the latest world state (based on the blockchain's head). For every transaction, the following steps are executed. The transaction is validated again according to the steps defined previously in Section 4.2.3, because the world state including account's available Gas could have changed in the meanwhile.

The transaction sender's nonce is incremented by one. Then, the upfront (standard) cost of executing the transaction is computed, and is deducted from the sender's Gas (crypto) balance using a GasPrice (0 for permissioned). After the default transaction cost is deducted from the transaction's gas limit, a certain amount of *Execution Gas* is left that may be used during the transaction execution. In fact, this Execution Gas is used to create a *Message Frame*, which is added to the top of the (empty) EVM stack. As explained in Section 3.2.5, this stack-based architecture is used to keep track of the operations to be performed. Next, the EVM keeps executing the message on top of the stack, until the stack is completely empty. There may be various reasons for a transaction to be aborted during runtime, such as running out of execution gas, or calling a `Revert()`, `Assert()` or `Require()` from a smart contract. If the transaction was executed successfully, the world state's view is updated. These are changes not propagated to the underlying storage (yet). Finally, the actual Gas that was used during the execution is deducted from the sender's account using the gas price, and is awarded to the address defined by the node.

At last, the block's header is updated with values resulting from the execution and the transaction pool is unlocked. The header update is the main reason that transactions must be executed during the block creation process: information such as a hash of the included transactions, a hash of the transaction results (receipts), the total gas used, and more. In addition, if a node includes an invalid transaction, the entire block is invalidated as will be explained later, highlighting the importance of only including valid transactions.

The block itself only contains the block's header and a list of transactions in its body. This process is shown in Figure 4.16. After this process is finished, the block is returned back to the consensus protocol, which is responsible for the distribution of the block and reaching consensus on it.

4.2.6 Phase 6: Importing a new block to the ledger

As briefly mentioned in the previous phase, the consensus protocol is responsible for achieving consensus on the block and distributing it to the other participants. No matter if a block was created locally or received by a different participant, the importing process of the block is equivalent.

The entire importing process is synchronized, meaning only one thread may import a block at the same time. First, a header field is extracted from the block that is unique for each consensus protocol. This header field is used to perform validation rules that are specific to the configured consensus protocol. Some examples the reader may think of are (1) ensuring the block was created by a designated validator, (2) the block might require enough seals of participants, or (3) for Proof of Work, it should have a valid hash matching the network's difficulty.

If the block's header was confirmed to be valid with respect to the consensus protocol, the block's parent is retrieved from the header. Then, the ledger's world state is computed (or rather: derived) to match the parent. For every transaction in the newly received block, it is processed in the same way as shown in the previous phase (i.e., validation steps, execution by EVM). Even if only one transaction in the block is invalid, the entire block is discarded. A view of the world state is updated after every transaction execution. Finally, after all transactions were executed successfully, the updates to the world state are persisted to storage. As a safety measure, the world state's hash is obtained and compared with the hash in the block, to ensure that the world state matches that of the block creator.

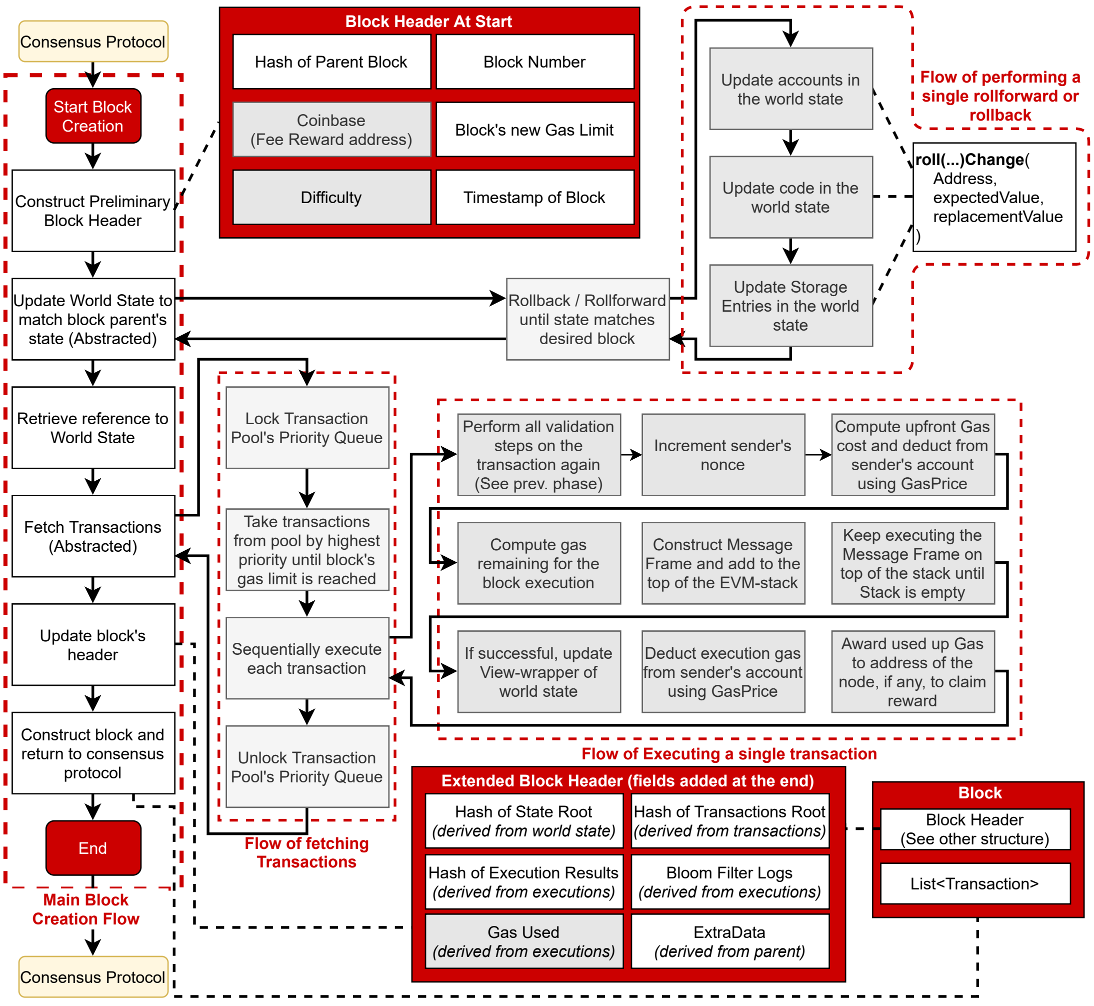

Consensus Protocol
Start Block Creation
Construct Preliminary Block Header
Update World State to match block parent's state (Abstracted)
Retrieve reference to World State
Fetch Transactions (Abstracted)
Update block's header
Construct block and return to consensus protocol
End
Main Block Creation Flow
Consensus Protocol
Block Header At Start
Hash of Parent Block
Block Number
Coinbase (Fee Reward address)
Block's new Gas Limit
Difficulty
Timestamp of Block
Rollback / Rollforward until state matches desired block
Update accounts in the world state
Update code in the world state
Update Storage Entries in the world state
Flow of performing a single rollforward or rollback
roll(...)Change(Address, expectedValue, replacementValue)
Lock Transaction Pool's Priority Queue
Take transactions from pool by highest priority until block's gas limit is reached
Sequentially execute each transaction
Unlock Transaction Pool's Priority Queue
Flow of fetching Transactions
Perform all validation steps on the transaction again (See prev. phase)
Increment sender's nonce
Compute upfront Gas cost and deduct from sender's account using GasPrice
Compute gas remaining for the block execution
Construct Message Frame and add to the top of the EVM-stack
Keep executing the Message Frame on top of the stack until Stack is empty
If successful, update View-wrapper of world state
Deduct execution gas from sender's account using GasPrice
Award used up Gas to address of the node, if any, to claim reward
Flow of Executing a single transaction
Extended Block Header (fields added at the end)
Hash of State Root (derived from world state)
Hash of Transactions Root (derived from transactions)
Hash of Execution Results (derived from executions)
Bloom Filter Logs (derived from executions)
Gas Used (derived from executions)
ExtraData (derived from parent)
Block
Block Header (See other structure)
List<Transaction>

Fig. 4.16 The process of creating blocks in Besu

After the world state has been updated and verified to match the block creator's world state, the ledger's block storage should be updated next. To do so, addition to the blockchain is synchronized, i.e. locked. Strangely enough, the block's position in the blockchain is only now checked: the block may not be already present somewhere along the chain, i.e. a duplicate[13]. If after this check, the block is indeed unique, the block's parent hash must point to a block in the block store. In other words, the block does not connect to a "ghost" block, meaning the new block's parent is known and valid. The (now) valid block is added to the ledger in the key-value store (by writing a mapping of the block's hash to the block's data). If this block now has the highest block number, the blockchain's head is updated to point to the new block. Finally, all changes to the ledger's key-value store are committed as an atomic operation. Afterwards, both locks on the ledger's blockchain and block addition are removed. This last phase is illustrated in Figure 4.17 and marks the end of Besu's transaction flow.

4.3 Findings

4.3.1 Transaction Flow Design

In this chapter, we have seen how transactions are created, executed, validated, and appended to the ledger by using the source code of both frameworks as source of truth. We saw that the transaction flow of both frameworks are fundamentally

[13]This includes checking for duplicates at the head, or further back in the chain.

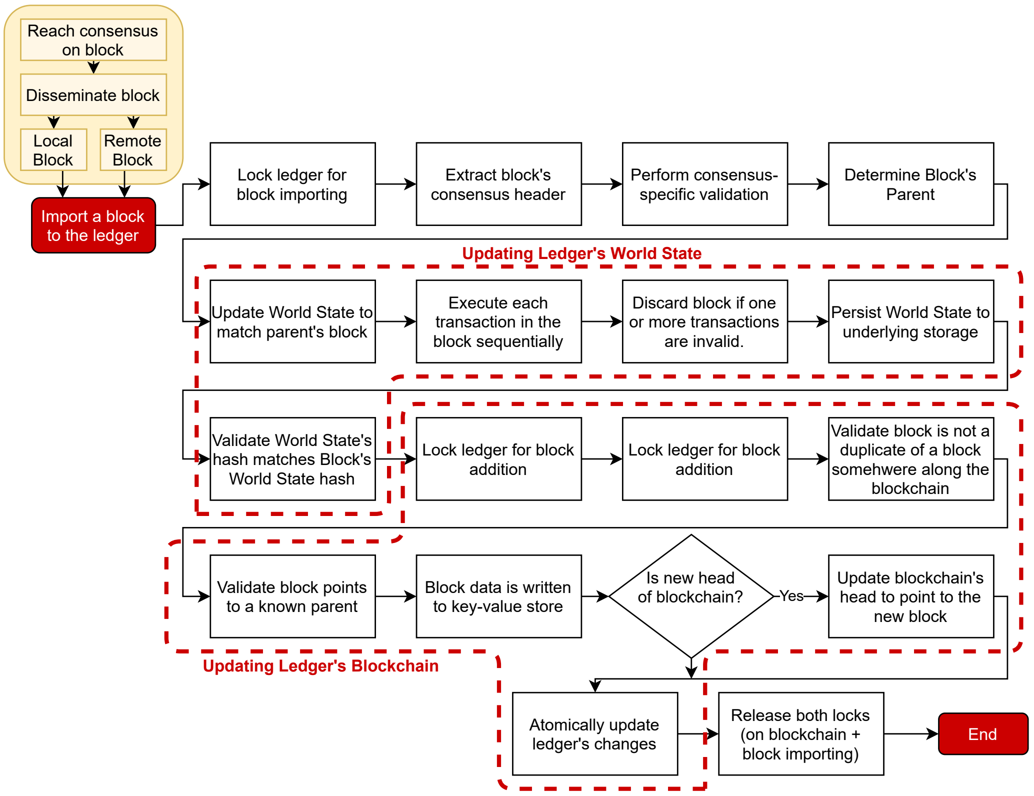

Reach consensus on block
Disseminate block
Local Block
Remote Block
Import a block to the ledger
Lock ledger for block importing
Extract block's consensus header
Perform consensus-specific validation
Determine Block's Parent
Updating Ledger's World State
Update World State to match parent's block
Execute each transaction in the block sequentially
Discard block if one or more transactions are invalid.
Persist World State to underlying storage
Validate World State's hash matches Block's World State hash
Lock ledger for block addition
Lock ledger for block addition
Validate block is not a duplicate of a block somewhere along the blockchain
Validate block points to a known parent
Block data is written to key-value store
Is new head of blockchain?
Yes
Update blockchain's head to point to the new block
Updating Ledger's Blockchain
Atomically update ledger's changes
Release both locks (on blockchain + block importing)
End

Fig. 4.17 Importing a block into Besu's ledger

different in a major way: Fabric follows the *Execute-Order-Validate* architecture, whereas Besu has a more traditional *Order-Execute* architecture. The design choices and implications of this transaction flow are compared next.

Actor Responsibility

For Fabric, the Execute-Order-Validate architecture results in the actor having a lot of responsibility. The actor not only needs to submit the transaction proposal to (multiple) peers, but it also needs to wait and collect their replies, validate them, combine these in a transaction and submit this transaction to the ordering service. Although all of these actions may be performed by an SDK, this still results in many operations being executed on the device. In Besu, the responsibility of a node is much less: namely to construct and send a transaction to any node in the network. In contrast to Fabric, this is mostly *Fire and Forget*, as the responsibility of executing and validating the transaction lies predominantly with the network, rather than the actor. In practical terms, this means that Besu might be a better alternative when an actor needs to be an extremely thin client on which computations are expensive, such as an IoT-device. Yet, it should be worth mentioning that this *could* be achieved with slightly more effort in Fabric as well: by having a thin client fire-and-forget a transaction proposal to a server that invokes the SDK on the IoT device's behalf, i.e. a delegation pattern. Then, this remote server would be responsible for discovering endorsers, sending the transaction proposal, fetching the endorsements, validating them, and forwarding the resulting transaction to the ordering service. Note, however, that this functionality is not supported by the framework. It would mean developing a very light server that receives a request and invokes the SDK.

Transaction Ordering

A fundamental difference regards the process of selecting and ordering transactions. We saw that transactions in Fabric are submitted to the ordering service, which establishes a total ordering of transactions. This order is established by a *group* of decentralized nodes, i.e. all OSNs together decide on the ordering. On the contrary, a block creator in Besu is allowed to order the transactions in the block it creates in any way it wants to on an *individual* basis. In fact, the source code itself revealed that a transaction submitted directly to a node by an actor always take precedence over a transaction received peer-to-peer from another node. Nothing prevents a peer from only including transactions in the next block that are beneficial to itself. Although this is probably an extreme case and does not happen a lot in practice, it is technically allowed.

Transaction Execution and Simulation

As hinted to earlier, the transaction flow has major implications on the stage at which transactions are executed. In Besu, every transaction is first simulated by the peer while creating a block. Then, every single node in the network executes the smart contract logic again when receiving this block. This results in a lot of redundant executions. In Fabric, a transaction is only executed by a number of peers marked as endorser by the actor, as dictated by the endorsement policy. Although it is possible in Fabric that every organization in the network must endorse a transaction, this is often undesirable[14] and only a subset of participants are required. As a consequence, a Fabric transaction, as recorded on the ledger, does not denote a function *execution*, but the *results* of that execution in terms of a read set and write set. This makes Fabric's transaction execution a lot more efficient, than Besu's. Yet, this requires more initial effort and understanding to come up with an endorsement policy that is sufficiently secure[15]. Both frameworks simulate the transaction execution in memory, at some point in time: Fabric during the endorsing process, and Besu during the block creation. These simulations by both frameworks are performed in-memory without persisting the changes to the data store, such that they do not need to be rolled back afterwards.

Comparison Overview on Transaction Flow Design

Transaction Flow Design		
Concept or Feature	**Hyperledger Fabric**	**Hyperledger Besu**
Transaction Flow Architecture	Execute-Order-Validate.	Order-Execute.
Actor Responsibility	Requesting, collecting, and validating endorsements, Constructing and sending a transaction.	Constructing and sending a transaction (Fire and Forget).
Transaction Ordering	Decentralized nodes determine as a group what transactions are included and their ordering.	Block Creator determines on an individual basis what transactions are included in a block and their ordering (using a local priority queue).
Transaction Executors	Subset of peers called Endorsers.	All peers execute transactions.
Transaction Execution	Only before block creation.	During block creation and during commit.
Transaction Simulation Technique	In-memory without persisting	In-memory without persisting

Table 4.1 Overview of differences between Fabric and Besu on Transaction Flow Design

4.3.2 Guaranteeing Ledger Integrity

The previous paragraph already explained that Fabric's transaction execution is much more efficient than Besu's when not all endorsers are required by the endorsement policy. However, this efficiency comes at a cost of additional risk to the ledger's integrity. This was explained in much depth by Soelman et al. [86]. In essence, transactions in a block are only validated in terms of an up-to-date *readset* (by the MVCC), whether the signatures are correct, and whether the endorsement policy is satisfied (by the VSCC). There are no safeguards in place that requires every peer to validate whether the *writeset* was properly generated by a chaincode. Technically, a set of colluding nodes that can together satisfy the endorsement policy are able to craft a transaction proposal and writeset together. This writeset could make any changes to

[14]When one organization refuses to endorse a transaction or becomes unavailable, not a single new transaction would be considered valid.
[15]The ledger is only as secure as the endorsement policy, as explained in the next paragraph.

the ledgers of all nodes in the network, e.g. by deleting all keys or by changing ownership of an asset that the colluders do not own. The colluders can "endorse" such a proposal by signing an *illegal* endorsement, i.e. one that was not properly derived from honest transaction execution. They can submit this transaction to the ordering service, which results in all peers in the channel blindly implementing the illegal changes. Although this might seem like a big problem, it simply means the endorsement policy should require as many endorsements as the network considers sufficiently safe.

Although Besu's transaction flow is less efficient and requires all nodes to execute every single transaction, this is Besu's main technique to guarantee ledger integrity. Namely, it is technically and by definition *impossible* for an actor to make *illegal changes* to an honest node's ledger, i.e. a change to the key-value store that was not the result of a proper transaction execution. In other words, a node never makes illegal changes, because it only accepts changes that it executed and generated itself (and thus agrees with). In fact, when a Besu node receives a transaction that tries to perform a disallowed operation, it simply discards it. Then, if there are other nodes that *do* consider the transaction as valid and perform ledger changes, it means a *hard fork* of the blockchain occurs. This means that the blockchain splits into *two*. Nodes that consider the transaction as invalid will form their own linked list of blocks, whereas nodes that accepted the transaction (and its changes) maintain their own linked list of blocks. Thus, if a group of participants try to "cheat in" a disallowed modification of the ledger, only the colluders will "pretend" it is accepted. The rest of the network that discarded the transaction as invalid will continue operating and creating blocks on the other fork that does not contain the modification by that transaction. Because new blocks also contain a hash of the state root, blocks submitted by dishonest nodes are discarded immediately by honest nodes. Thus, the ledger of the two groups evolve separately. For a participant without its own honest node, there is no way to know which nodes are honest or dishonest and the only way to solve this, is by interacting with the network majority or force the dishonest nodes to resync.

Preventing Double Spends

An important problem the transaction flow aims to solve is to prevent the well-known double spend problem, in which the effects of a transaction on the same state are applied twice [30]. Fabric prevents this using a process called MVCC, as explained in Section 4.1.8. It is crucial this step is performed, as the transactions are executed *before* ordering, rather than *during* the commit phase. Essentially, a transaction is only valid if the version included in a transaction's readset still matches the version number written for that key on the ledger. Thus, if a version of at least one of the possibly many reads have changed, the writeset was generated using a different state, and is thus considered invalid. The MVCC is performed on each peer during the block commit phase. Besu does not require such an MVCC-check, since transactions are executed and validated *during* the commit time, after they were included in a block. The ledger state is thus always up-to-date when transactions are executed. Yet, there is another problem that Besu must solve. Since ordering is performed locally by individual nodes, it may happen that an actor submits two consecutive transactions, but are included in a block in reverse order. To solve this, as elaborated in Section 4.2.3, every transaction is linked to a transaction creator with its own nonce. The creator's nonce is incremented for every transaction he or she makes. If a transaction has a *lower* nonce than stored on the ledger (for that creator), it means that the transaction should have been performed *before* a later transaction of that same account[16]. Thus, by discarding transactions with a nonce *lower* than the account's nonce, the protocol ensures the ordering of transactions are faithful to the order the *actor* submitted them.

[16]This also prevents a possible *replay attack*, in which an adversary catches the transaction and submits the same transaction (such as a value transfer) multiple times.

Comparison Overview on Guaranteeing Ledger Integrity

Guaranteeing Ledger Integrity		
Concept or Feature	**Hyperledger Fabric**	**Hyperledger Besu**
Ledger Integrity Guarantee	None, based on authority of endorsers as defined by endorsement policy.	Values are guaranteed to be properly derived from execution. All nodes execute the transaction, and those that disagree create a soft fork.
Preventing Double Spends	MVCC using a versioned read set	Execution during block commit on latest ledger state, in combination with account-based nonces

Table 4.2 Overview of differences between Fabric and Besu on Guaranteeing Ledger Integrity

4.3.3 Communication and Connectivity

Actor Connectivity

In terms of communication, there are also some large differences between the frameworks to highlight. First, consider the actor's communication with the network. In Besu, an actor only needs to have the address of any node in the network, provided access control allows it. An actor can retrieve blocks, query the ledger, simulate a transaction without submitting it, obtain peer metrics, submit a transaction and many other operations all by communicating with a single node. In Fabric, an actor communicates with a lot more entities. Not only does it need to know or discover the address of every single endorser that is required to satisfy the endorsement policy, the actor also needs to connect to OSNs. Although this shows that an actor has to make relatively many connections, it also implies an actor in Fabric requires good network connectivity. In case a Fabric network is geographically distributed and connected using the internet, sufficient connectivity for the actor would be important. For Besu, this is not the case, as a local connection between the actor and a node is sufficient, as long as the node itself is connected to the rest of the Besu network (e.g. using internet). Again, this may be a consideration for IoT devices.

Block Delivery Connectivity

Another type of connectivity relates to how blocks are distributed throughout the network. In Fabric, blocks are created by the ordering service which is *logically centralized* but may be *physically decentralized*. These blocks are delivered to all subscribers using the publish/subscribe pattern. These are implemented using connections between a single OSN and a single peer that typically remains open indefinitely. Every organization should have (at least) one peer subscribed to an OSN. Within a single organization, however, blocks are propagated through a peer-to-peer gossip protocol. Nodes in Besu maintain a list of other nodes that are connected to the network, which are automatically populated through peer-to-peer discovery. The maximum amount of (active) nodes that are maintained in memory is configurable, but by default set to a maximum of 25. Any node in a Besu network may create a block, as decided by the consensus protocol. Then, if any of these nodes have created a block, this block is transmitted to all other nodes in this list. If this block was deemed valid, a receiving node propagates the block further to other peers.

Block Delivery Mechanism

Blocks in Fabric are not immediately broadcasted to all subscribers, but only after the block buffer of an OSN or peer has exceeded a *number-of-blocks threshold*, or if a *time threshold* has been exceeded from when the first block entered the buffer. Then, the entire buffer is sent out as a batch to all subscribers. Batching techniques typically improve throughput by reducing communication overhead, although this goes hand in hand with an increase in latency. Since the gossip protocol only pushes blocks to a maximum number of random peers, some peers may be unlucky and do not receive a block. Thus, each node occasionally pulls blocks as well from either the ordering service or random peers of the same organization. Since blocks in Besu are typically created anywhere in the decentralized network, blocks are sent individually and not as a batch. Besu also implements pulling, but in a slightly different way. Every Besu node *synchronizes* with exactly one other node taken from the list of discovered nodes mentioned earlier. This synchronization corresponds to a series of steps in which the synced node's headers are downloaded and used to request missing blocks. In essence, it means a node *is trailing* the blockchain of another node.

Comparison Overview on Communication and Connectivity

Communication and Connectivity		
Concept or Feature	**Hyperledger Fabric**	**Hyperledger Besu**
Actor Connectivity	Connects to all required endorsers of other organizations, and at least one OSN.	Connects to only a single node or EthSigner proxy for submitting transactions.
Block Delivery Connectivity	Ordering Service delivers to all subscribed peers, at least one per organization.	Block creator broadcasts to all connected nodes (by default a maximum of 25).
Block Delivery Push Technique	Ordering service pushes batch of blocks after creation, or peers within organization push batch of blocks through gossip after receiving.	Block creator and peers push single block through gossip after creating or receiving.
Block Delivery Pull Technique	Nodes frequently pull batches from the ordering service or from peers using gossip.	Nodes only sequentially pull blocks one-by-one while syncing at start-up from exactly 1 other peer (the sync target).

Table 4.3 Overview of differences between Fabric and Besu on Communication and Connectivity

4.3.4 Transaction Flow Functionality

Rolling back the world state to a previous block

The design of the transaction flow has several implications related to functionality and supported operations. Rolling back to a previous block is one such functionality. A Besu node must have this feature by definition. Since a Besu blockchain might fork into two heads, chances are another chain becomes longer[17]. Then, the node needs to roll back to previous blocks. Besu performs these operations by storing triplets after the execution of each block in the form of `(key, old, new)`. This allows a node to rollback to previous blocks without having to execute the transactions in a block again. Although this functionality was created for the blockchain to function properly, this functionality is also exposed to actors. By calling the `eth_call` method, an actor may simulate a transaction or read a value on a prior state by specifying a block number. In Fabric, old values are not stored. Therefore, rolling back to the state of previous blocks is unsupported, unless a node starts syncing from the genesis block again.

Determinism of Smart Contracts

Naturally, it is important to eventually reach consensus on the state of the blockchain. Otherwise, (all) participants have different states, resulting in a disagreement on what ledger information is true and what not. Besu enforces this by requiring all smart contracts to be deterministic. This means that as long as the world state and arguments are equivalent, executing a smart contract always results in the same output. Because of the determinism of smart contracts, if all individual nodes execute all transactions to update their own state, all nodes always apply the same changes. Although this might seem obvious, this severely restricts the types of operations a smart contract may perform. Some examples include: (1) using System Time in a smart contract[18], (2) using a (pseudo) random number generator that cannot be predicted by the invoker, nor the block creator, or (3) getting information from a data source external to the blockchain, such as from a website's API[19] or from the node's file system. Because transactions in Fabric are already executed (endorsed) before a transaction

[17] As we see in the consensus protocol chapter, this depends on the consensus protocol used. Some consensus protocol with probabilistic finality use these rollbacks, whereas others with immediate block finality do not need to as blocks are appended forever.

[18] Although it is possible to get the block's creation time

[19] Distributed Applications overcome this by creating *Oracles*. Instead of fetching information from an external source, these are external sources that "bring" data to the ledger.

is ordered, chaincodes *may* be nondeterministic. A Fabric node is free to call any website's API, use the node's system time, or fetch information from its local file system. However, a disclaimer applies here: To prevent different world states, all collected endorsements must be consistent (in terms of read and write set). If the endorsements are inconsistent, the transaction is rejected. So although this allows for much greater flexibility in terms of smart contract functionality, it should be used with caution.

Turing Completeness and Termination

Section 2.3.1 already briefly described that a process that does not terminate could be very problematic for blockchains. According to Alan Turing's halting problem, it is generally impossible to tell if a program will terminate or not, before or during execution. A simple example could be smart contract `SC-A` that recursively calls smart contract `SC-B`, which calls `SC-A` again. Especially for Besu, this is a problem as transactions are executed after ordering: If a transaction *never* terminates, the next transaction will *never* be executed due to their sequential ordering. Thus, if a transaction seems to take a long time, when do we kill it? A timeout in seconds does not work, as execution time differs on different hardware, which might cause a transaction to be killed on one node, but successfully finished on another. Instead, each transaction has an upper limit of the amount of Gas it might use as fuel for each operation. After Gas runs out, the transaction is killed. This results in all nodes killing the execution of a transaction at the exact point of execution. In Fabric, endorsers also kill transactions to prevent execution to infinity. However, since nondeterminism is not a problem, a node may decide to abort a transaction whenever it wants to. In practice, this happens through a maximum amount of seconds defined by an endorser's local configuration file. Since transaction execution by both frameworks are Turing Complete but only restricted by time (in terms of seconds or operations/gas), Fabric and Besu are both said to be Quasi Turing Complete [97, 3].

Comparison Overview on Transaction Flow Functionality

Transaction Flow Functionality		
Concept or Feature	**Hyperledger Fabric**	**Hyperledger Besu**
Rollbacks to previous blocks	Unsupported, requires recomputing from the genesis block.	Supported, may be used by actors to simulate transactions or read values at an older block.
Determinism of Smart Contracts	Smart Contracts are allowed to be undeterministic.	Smart Contracts must always be deterministic.
Interaction External to Network	Allowed through any supported programming language or library. If endorsements are inconsistent, transaction is invalid.	Disallowed, as it can result in inconsistent ledger state due to transaction execution.
Turing Completeness	Quasi Turing Complete, restricted by time in seconds.	Quasi Turing Complete, restricted by time in number of operations and gas.
Moment to kill transaction	The endorser's local configuration defines the number of seconds after which to kill the transaction.	The transaction's gas limit is set by the submitter of the transaction.

Table 4.4 Overview of differences between Fabric and Besu on Transaction Flow Functionality

4.3.5 Safety Properties

A *safety property* is a strict guarantee that may not be broken. If such a safety property applies to a framework and *would* be broken, things will go wrong and critical exceptions will occur. Various safety properties are described next. For each of these, it is mentioned whether the safety property is required and satisfied by that framework or if it is not required due to its transaction flow. The first five were identified by Androulaki et al. [3].

Agreement

First of all, there could be **Agreement** on the sequence numbers of blocks: If blocks $B1$ with sequence number $S1$ and block $B2$ with sequence number $S2$ exist, and the sequence numbers are equivalent (i.e. $S1 == S2$), then it must mean that both blocks are equivalent, i.e. $B1 == B2$. In other words, no two different blocks may have the same sequence number. This property is satisfied in Fabric by its ordering service, since it never creates a fork. Besu does not satisfy the agreement property, as forks with the same block number are allowed. Yet, it depends on the consensus protocol whether these canonical blocks are actually created in practice.

Hash Chain Integrity

Androulaki et al. calls a second safety property **Hash Chain Integrity**: If block $B1$ with sequence $S1$ is created and block $B2$ with sequence $S1+1$, then it must hold true that the block $B2$ contains a hash of block $B1$. This essentially means that each block must include a hash of the block with the previous sequence number (with the exception of the genesis block, of course). In other words, each block is ordered such that it is a child of exactly one previous block. Both Fabric and Besu satisfy this property, as each block contains a hash to its parent with the previous sequence number.

No Skipping

Thirdly, **No Skipping** of blocks could be a safety property. This means that if a peer has last received a block with some sequence number $s1$, then the next block must have sequence number $s1+1$. In other words, peers should only store (and receive) blocks in the exact order as defined by their sequence number. Consequently, this property dictates that it is not allowed for a peer to have gaps of missing blocks on its ledger. This restricts the block delivery, as it always needs to deliver blocks in their natural order: Blocks delivered in any random non-consecutive order should be discarded. Fabric satisfies this property through the ordering service and through gossip: the deliverer maintains the last block number each subscriber knows of, and always starts delivering blocks from lowest to highest sequence number. Besu also satisfies this property, but No Skipping is enforced by the receiver (as opposed to the deliverer). During the syncing process, a node requests blocks one-by-one from another (target) peer. After syncing has finished, blocks may be delivered in any order, but the receiving node only accepts blocks of which it has its parent in storage.

No Creation

A more trivial safety property by Androulaki et al. is that **No Creation** is allowed by the block creator. This means that every single transaction added to a block, must have been broadcast earlier by an actor. Besu and Fabric both satisfy this property, since neither nodes nor Fabric's ordering service create new transactions (i.e., the actors create them). Moreover, this is further enforced by requiring individual transactions to always remain signed by the actor, while they are included in blocks.

Validity

The last of the five safety properties identified by Androulaki et al. regards **Validity**. Validity means that every valid transaction that an actor submits, must *eventually* be included in a block. Fabric's ordering service satisfies this requirement by using consensus protocols that guarantee this. Besu, however, does not satisfy this property. Recall that transactions in Besu may be ordered differently by nodes and that the transaction pool has a limited capacity. Thus, a transaction may be dropped if the transaction pool reaches its maximum capacity. Therefore, it is not guaranteed that a transaction is eventually included in a block, after an actor has submitted it to a Besu node. In these extreme cases, the ledger would not be changed and one would have to resubmit the transaction once the transaction pool is no longer full.

Fairness

Section 2.1 briefly mentioned **Fairness** as a DLT-property, meaning all transactions are included in the blockchain in the same order as they were created or submitted. More formally: Given Transaction `T1` submitted at time `t`, and Transaction `T2` submitted at time `t+1`, then `T1` should always appear earlier in the blockchain than `T2`. For Fabric, this property is *not* enforced by the ordering service. In fact, if transaction $T1$ at time t and transaction $T2$ at time $t+1$ are submitted to an OSN with different latency (due to connectivity speed or hardware delay), any of them could appear first in a block. Likewise, the same applies to Besu, which also does not satisfy this property. This should be obvious for Besu, since each node establishes its own transaction order.

Comparison Overview on Safety Properties

Safety Properties		
Concept or Feature	**Hyperledger Fabric**	**Hyperledger Besu**
Agreement	Satisfied.	Not Satisfied, soft forks create multiple blocks with the same sequence number.
Hash Chain Integrity	Satisfied.	Satisfied.
No Skipping	Satisfied, enforced by ordering service (deliverer).	Satisfied, enforced by receiving node.
No Creation	Satisfied.	Satisfied.
Validity	Satisfied.	Not Satisfied, transaction may be dropped when transaction pool is full.
Fairness	Not Satisfied.	Not Satisfied.

Table 4.5 Overview of differences between Fabric and Besu on Safety Properties

Chapter 5

Consensus Protocols

Fabric and Besu both support *modular* consensus protocols, meaning the network configurator is able to choose a consensus protocol out of multiple options. As we have seen in the previous chapter, the consensus protocol is deeply embedded in the transaction flow of both frameworks. This transaction flow revealed that the output of both consensus protocols are similar, namely to produce blocks. Yet, we also saw that Fabric only tries to create a new block when triggered by a new transaction (or its timer), whereas Besu's consensus protocol continuously creates (possibly empty) blocks. This chapter aims to answer the sub-research question "***How do the modular consensus protocols work, and what are their differences?***" Again, this chapter follows the same structure as the previous two: First, Fabric's consensus protocol is explained in Section 5.1. Then, Besu's consensus protocols are discussed in Section 5.2. And lastly, to conclude, their differences are highlighted in Section 5.3.

5.1 Reaching consensus in Fabric

The main goal of Fabric's consensus algorithm is to create blocks that are agreed upon by all OSNs, even in case some OSNs might fail. There are three different consensus protocols that Fabric ships with, namely Raft, Kafka and Solo. Of these, only the Raft consensus protocol is actually encouraged by the documentation: Kafka and Solo are both deprecated.

Solo is the simplest consensus protocol of the three and only supports running a single OSN. Solo is implemented by initializing a *Golang channel* (sort of a local queue) on the OSN. For every transaction a OSN receives, the transaction is put in this queue. Then, an infinite loop is continuously reading transactions from this queue and adds them to the next block until the block is cut. The reading order of this local channel is the exact order in which transactions are added to blocks. Because the Solo consensus protocol only supports running a single OSN at a time, it is naturally easy for the node to reach "consensus" with itself: The node may order transactions any way it wants to, as it does not have to negotiate and communicate this with other nodes. For obvious reasons related to availability and fault tolerance, Solo was only to be used for testing and development purposes (before it was deprecated). Now, Raft can be configured for a single node ordering service as well, eliminating the need for Solo.

Prior to version v1.4 of Fabric, all OSNs used Kafka and ZooKeeper for ordering transactions into blocks. Kafka is based on a leader-follower principle, in which only the leader is allowed to perform writes. These writes are then copied to all followers. This same leader-follower concept is implemented by Raft. In fact, the documentation indicates that a user of Fabric does not notice any functional differences between the two and provides five arguments for using Raft over Kafka[1]. First, Raft is embedded, deployed, and managed when starting an OSN, whereas Kafka would have to be deployed and managed as independent instances. Secondly, a cluster of Kafka nodes would have to be deployed by a *single* organization

[1] https://hyperledger-fabric.readthedocs.io/en/release-2.2/orderer/ordering_service.html#raft

due to practical limitations, whereas a Raft cluster may consist of nodes of *different* organizations. Thirdly, support is provided for Raft as its implementation is embedded in the source code, whereas support for Kafka is handled through the Apache Software Foundation. Fourth, Kafka requires a (central) administrator to manage the cluster, whereas a Raft cluster is automatically managed through a channel's configuration. Lastly, the documentation mentions Raft could be a first step towards reaching BFT, although both Kafka and Raft are only exhibiting Crash Fault Tolerance (CFT). Yet, it is unclear how Raft could move towards BFT as an argumentation of this claim is not provided.

Although Fabric only endorses using one of the three modular consensus protocols, people are free to implement their own consensus protocol. One such consensus protocol is Hedera's Hashgraph. Recall that its basic concepts were discussed in Section 2.1. Essentially, Hedera created this "consensus module" by forking the Fabric project[2] and adapting it to work with the Hedera Consensus Service[3]. This takes the shape of a *Consensus-as-a-Service* model, in which consensus is outsourced to nodes run by Hedera. These nodes are governed by a council of 39 members to achieve decentralization while reducing the risk of colluding members by maintaining strict entry conditions [8]. Users of this service would have to pay a transaction fee of $0.0001 per message to order. Their forked Fabric project is part of `hyperledger-labs`, meaning it is not officially admitted to the Hyperledger consortium (yet). In addition, their repository is updated much less frequently and is lagging behind on the latest updates to Fabric. Furthermore, Hedera is considered highly experimental. Because of these reasons, the Hedera consensus protocol will not be considered further for this thesis.

Since Raft is the only officially supported consensus protocol for Fabric, the remainder of this section focuses on elaborating Raft, how it works within the Fabric framework and its implications.

5.1.1 Reasoning behind Raft

Prior to Raft, many consensus algorithms were based on Paxos by Lamport [58, 1]. Examples include Google's cluster manager *Borg* and *BigTable*. *ZooKeeper* is a key-value store based on *Zab*, also is heavily inspired by Paxos [1]. Because integrating Zab's Paxos implementation into applications was considered quite complex, many applications used ZooKeeper for consensus purposes including (but not limited to) *Kafka*, *Hadoop*, *Cassandra* and *Neo4j* [1].

Despite its popularity, Ongaro and Ousterhout, the designers of Raft, saw significant drawbacks of Paxos [72]. First, they considered Paxos "*exceptionally difficult to understand*". In essence, Paxos is used to reach agreement on a *single* decision. In order to establish a *log* or history of these decisions, multiple instances of the Paxos protocol need to be combined, making the algorithm difficult to grasp by even experienced researchers and scholars. Secondly, the authors observed that there did not exist a widely accepted implementation of Paxos at the time. The original description of Paxos would have been too abstract, making it often very complex to implement the protocol in practice. This claim is strengthened with a quote from the developers of Chubby, a distributed lock manager based on Paxos: "*There are significant gaps between the description of the Paxos algorithm and the needs of a real-world system. [...] the final system will be based on an unproven protocol*" [21].

Due to a lack of a better alternative, Ongaro et al. designed their own consensus protocol: *Raft*. Besides safety and efficiency, their primary goal was to make it understandable to the general public through problem decomposition and keeping *state* as simple as possible. In addition, Raft not only provides consensus on a *single* value (like Paxos), but also on the ordering (i.e. sequence or history) of those decisions.

Since the CAP theorem states one cannot have consistency, availability and partition tolerance all at the same time [17], Raft assumes *eventual consistency*. This means that there is no guarantee consensus is reached and stored on every instance (i.e. OSN) within a guaranteed time frame, but that this will happen at some point in the (near or distant) future.

[2]https://github.com/hyperledger-labs/pluggable-hcs
[3]https://hedera.com/consensus-service

Today, Raft has been implemented into Hashicorp's *Consul*, Hashicorp's *Nomad*, and *etcd*. Etcd is a key-value store similar to ZooKeeper. While etcd is mostly known for its use in *Kubernetes*, it is also used to implement the consensus protocol in Hyperledger Fabric.

Raft is especially interesting for Fabric, due to its various networking assumptions (or rather its tolerance thereof)[4], as identified by Koller [56]. First of all, Raft assumes *asynchronicity*. This entails that messages may not be received and handled immediately. In fact, messages may be delayed (and replied to) for an indefinite amount of time. Secondly, Raft assumes that network links are unreliable, i.e. packets may be lost, reordered, or duplicated. Even network partitions may occur. Thirdly, Raft falls under the category of CFT. This is an attribute of distributed systems that allows processes to crash and recover, as long as a majority of nodes are healthy. In other words, if a system should tolerate up to `f` crashes, then the total number of nodes in the network should be at least `2f+1`.

Although the CFT-attribute allows just under half of the processes to fail, the reader should be aware that *all* processes must be *honest*. As such, not a single process in Raft may be a malicious adversary. In fact, CFT is a weaker form of BFT. In BFT, one fault `f` requires a minimum of `3f+1` total processes to continue operations safely. More importantly, the faulty processes of BFT *are* allowed to be malicious adversaries trying to undermine the system, which is not the case in CFT. Since Raft is only CFT but not BFT, every single process must be honest. This is an important consideration, especially when the network consists of possibly distrusting members, as we will see later in Section 5.1.6.

5.1.2 Raft for realizing a Replicated State Machine

Ongaro et al. argue that consensus protocols are often used within the context of Replicated State Machines (RSM), while providing Chubby, ZooKeeper, Hadoop and Google File System as example. As shown in Figure 5.1, a RSM consists of a consensus module, a log, and a state machine. In short, the *Consensus Module* defines the rules and actions to perform on its own process, as well as the communication with other processes. The *Log* is a history of decisions, e.g. `SET idx=1 TO newvalue`. The values in the log may be *proposed*, meaning consensus has not been reached on that decision *yet* and may still be overwritten by another process. Alternatively, the log entries may be *committed*, meaning the decision has reached consensus and is final. The last component, the *State Machine*, could be any program on which the command/decision is to be executed. Whereas more traditional consensus algorithms like Paxos only try to provide a solution to realizing a consensus module, the Raft algorithm also describes the semantics of the log and state machine. Thus, the Raft algorithm prescribes how to realize a complete RSM, rather than just the consensus algorithm.

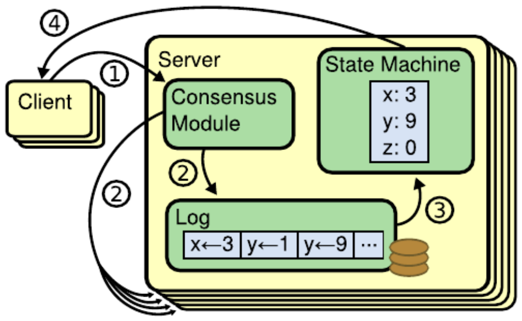
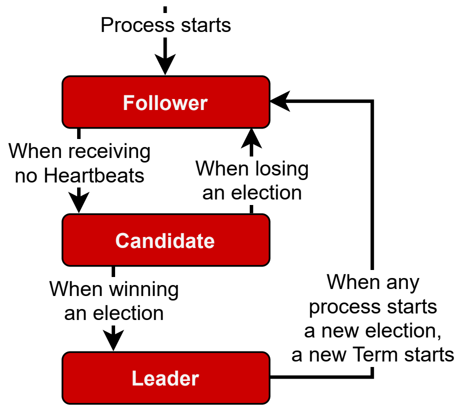


Fig. 5.1 Replicated State Machine [72]

Before going into detail on how Raft works and what it does, the reader needs to understand the *roles* a process may take. Raft takes a leader-follower approach. The *Leader* is a process with three major responsibilities. First, all writes of the client are routed to the leader which it appends to its local Log. Secondly, the leader continuously copies its own Log to all Followers. Thirdly, a leader periodically sends heartbeats to all followers, to let it know it is still operating. Most trivially, the *Follower* is a passive process that listens to decisions from the leader to append to its own log. If a Follower does not

[4]Multi-Paxos also makes the same three assumptions that are discussed next.

receive a leader's heartbeat for a (randomized) period of time, the Follower becomes a *Candidate*. A Candidate is a process that increments the Term (explained in the next paragraph) and tries to become the new leader. An overview of roles and their transitions is illustrated in Figure 5.2.


Process starts
Follower
When receiving no Heartbeats
When losing an election
Candidate
When winning an election
When any process starts a new election, a new Term starts
Leader
Fig. 5.2 Process Roles in Raft

Raft does not rely on the notion of continuous time, and as such, processes do not need to synchronize their System Times (or CPU clocks). Instead, Raft divides time up into abstract epochs which it calls *Terms*. Every time a follower does not receive the leader's heartbeats, it starts a new term. In practice, this term is a simple integer that monotonically increases every time a new election starts. Thus, starting a new election means a process' local term is incremented from `t` to `t+1`. If a leader was successfully elected, it will remain the leader for the entirety of the new term. In case no leader could be elected, as we will see in Section 5.1.3, the term ends without a leader, and a new election and term is started. Figure 5.3 illustrates this notion of Terms.

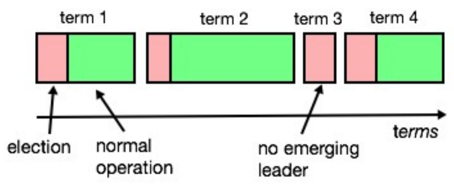


Fig. 5.3 Terms in Raft [72]

Now that Raft's roles and concept of terms have been explained, we can look at how changes are submitted to the RSM. These steps were also shown in Figure 5.1. As a first step, (1) the client's request is forwarded to the process with the leader role. If the client only knows of a follower, the follower should forward the request to the leader instead. Then, (2) the leader appends the client's command to the end of its local Log, and instructs all followers to do the same. At this point in time, the write operation is still a proposal and not final in any way. After the leader has received a confirmation of a majority of followers, (3) it commits the change by executing the state machine and (4) returning a response to the client. Followers are notified of the entry a leader committed lastly, by including its index in every heartbeat. The same process is illustrated in a sequence diagram in Figure 5.4.

5.1.3 Raft's Leader Election

As mentioned in the previous section, an election is started when a follower has not received a leader's heartbeat for a randomized interval. When the processes are initially started, everyone is a follower and an election is started at the end of

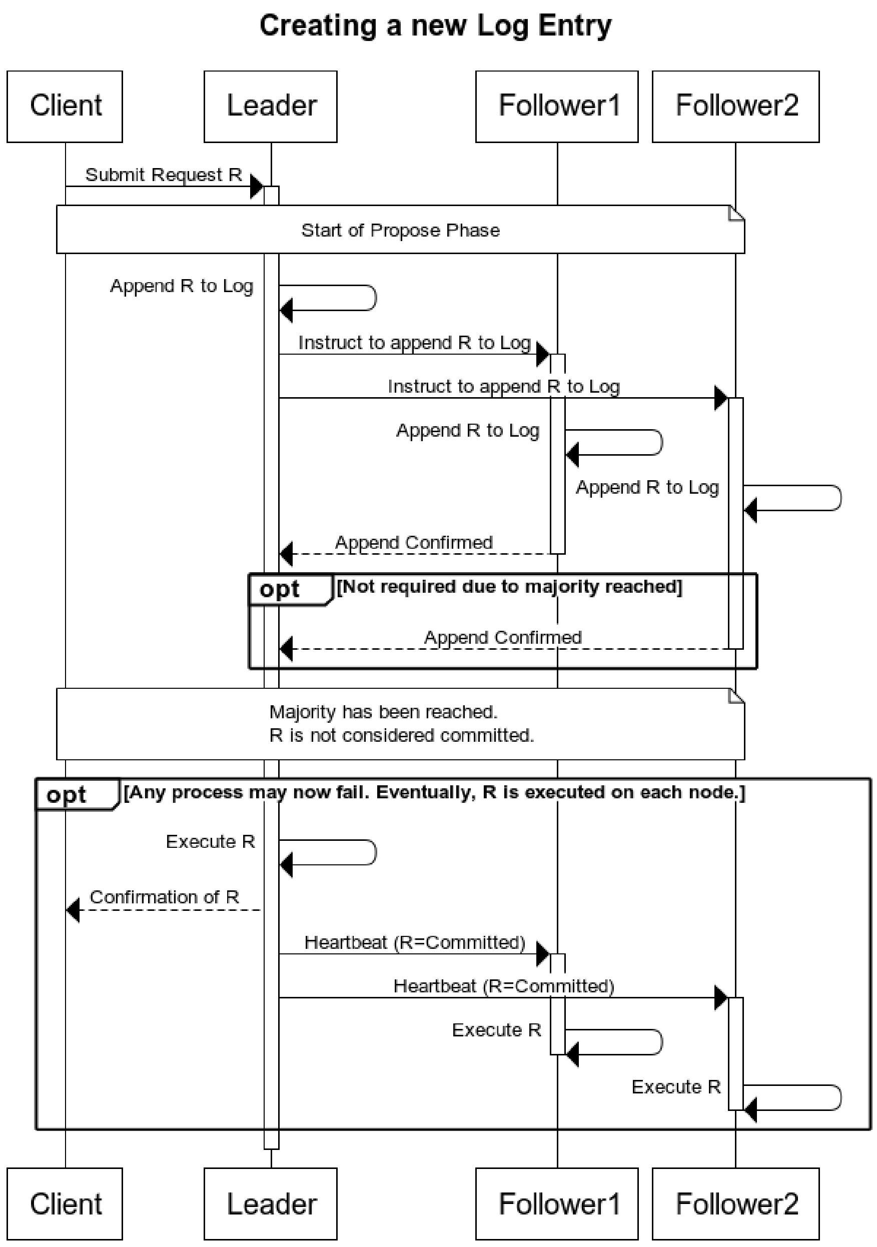


Fig. 5.4 Reaching Consensus on a new Entry in Raft

the first process that times out. An election is also started when a leader crashes, is overloaded, or its connectivity is gone, the follower whose timeout expires first starts the new term by becoming a candidate. Each candidate always votes for itself, and submits a `RequestVote` to all other processes. There are three different types of outcomes to an election, from the perspective of a process.

Three outcomes of an Election

Firstly, in case the candidate receives a majority vote, it promotes itself to a leader and starts sending leader heartbeats. Secondly, it could be the case that another follower became a candidate simultaneously and received more votes. Then, when the minority candidate receives the leader's heartbeat, it knows it has lost the election and becomes a follower again. A third outcome is when a candidate does not become a leader, and does not receive another election winner's heartbeat for some amount of time. This could have been caused by various reasons. For instance, a split vote could have occurred, in which two participants have the same amount of votes without reaching majority (such as a 2-2 split). Another example could be when another candidate was elected as leader, but crashed without sending the first heartbeat. If either of these conditions are true, a new election starts by increasing the term while trying to gather a majority of votes again. Because there exist terms in which not a single process is elected as leader, a new election might follow directly after. There does not exist an upper bound to the amount of time a process can be a leader, as long as all healthy followers are receiving its heartbeats.

Rainy Day: Majority of processes go offline

Now, consider the case in which a majority of processes go offline. We consider the case in which the leader stays healthy, or the case in which the leader goes offline as well. When the leader remains healthy, it will continue sending heartbeats and making changes to the ledger (because no other candidate emerges). However, because the leader does not receive a majority of confirmations on its changes, the changes it makes are only proposals and never committed. Alternatively, in case the leader itself is included in the majority of nodes that are offline, no candidate is able to gather a majority vote, and thus, the network is left without a leader. Either way, in both situations, the network becomes effectively nonfunctional as no changes will be final.

The election process visualized

Figure 5.5 provides an example of multiple leader election processes. These include sunny day scenarios, in which a leader is successfully elected. Rainy day scenarios are visualized as well, such as the case in which 1 or 2 nodes become unavailable. It also shows how the network becomes nonfunctional when too many nodes are unavailable[5].

5.1.4 Log Replication within Raft

The last piece of the puzzle concerns itself with how the Log Replication takes place. Section 5.1.2 and Figure 5.4 already explained how an entry is routed from a client to a leader, who writes the changes to (a majority of) followers, or at least tries to. After a leader has received a majority of confirmations, the change is committed. In this section, we look at what a *Commit* actually means, and how it relates to Terms.

First, we need to understand how a log entry is persisted by each process. Essentially, the log itself is an indexed list. Each log entry contains two pieces of information: (1) the Term at which it was written, and (2) the command or request of the client. The images by Ongaro et al. visualize this clearly, by giving each term a different color, as shown in Figure 5.6. In this image, we see the log of five different participants. This example shows that 3 followers are lagging behind on the leader, for instance, due to crashing, loss of connectivity, or due to temporarily being overloaded.

Recall that a log entry is committed once a leader has received a majority number of confirmations. Taking the example of Figure 5.6 and assuming that the leader received a confirmation from process #3 and #5, all log entries up to index 7 should be committed. The only operation a process has to perform to make this commit, is to update a global variable denoting the index of the highest committed entry. In this case, the commit index should be set to 7. Because the leader includes the last committed index and term in its heartbeats to all followers, each follower knows what values it is missing in case messages get lost due to whatever reason.

We already saw that a leader keeps operating, even when it can no longer connect to a majority of processes (although its changes cannot be committed). This may actually result in different processes with different log entries at the same index: A network partition may separate a leader from five followers. Then, one of these followers becomes the new leader. Now, the network has two (isolated) leaders. If both leaders still receive requests from clients (while the partition is in place), their log entries will eventually become inconsistent. An example is illustrated in Figure 5.7.

[5]The interested reader may also watch a visual animation at http://thesecretlivesofdata.com/raft/

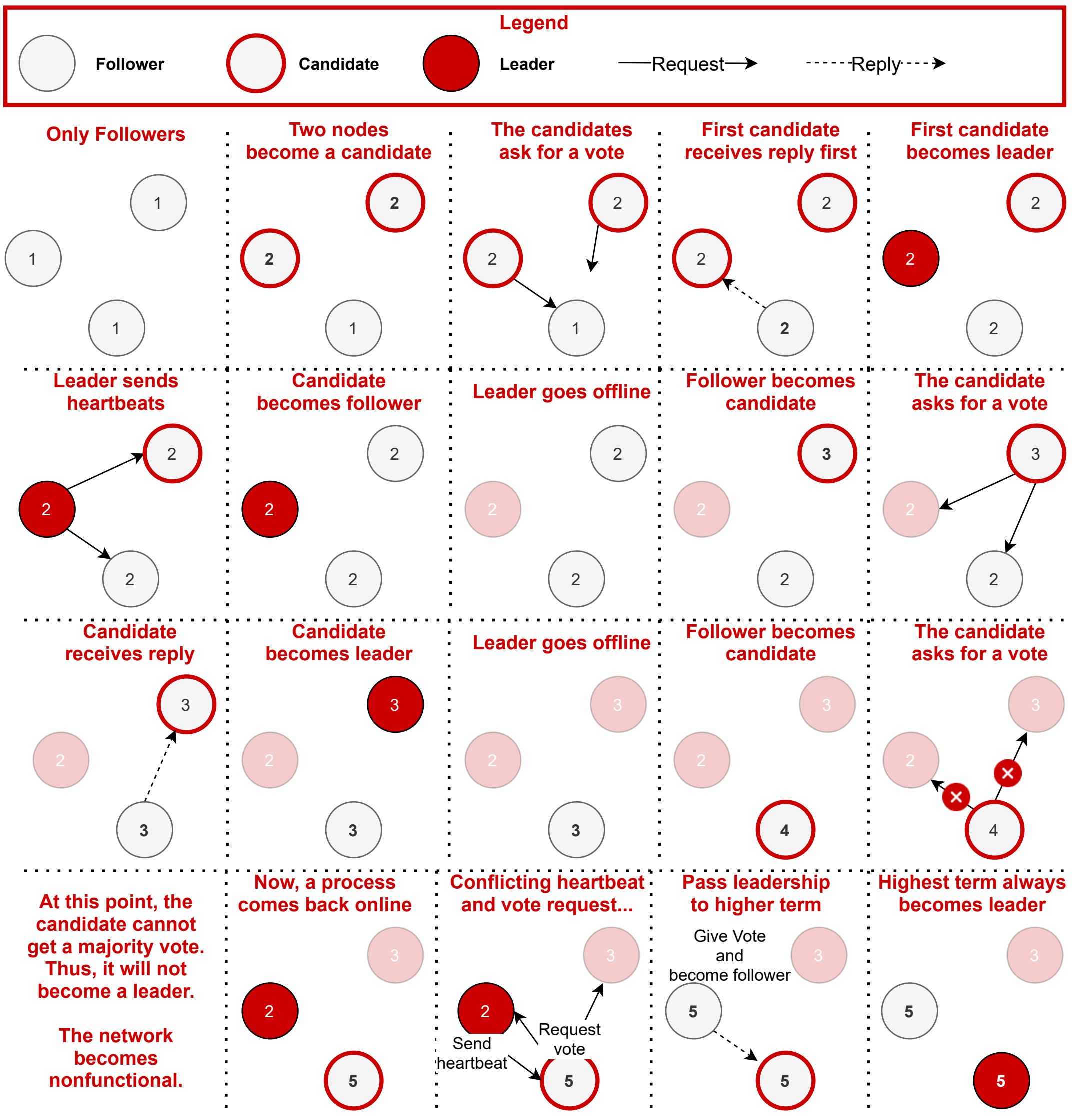

Legend
Follower
Candidate
Leader
Request
Reply
Only Followers
Two nodes become a candidate
The candidates ask for a vote
First candidate receives reply first
First candidate becomes leader
Leader sends heartbeats
Candidate becomes follower
Leader goes offline
Follower becomes candidate
The candidate asks for a vote
Candidate receives reply
Candidate becomes leader
Leader goes offline
Follower becomes candidate
The candidate asks for a vote
At this point, the candidate cannot get a majority vote. Thus, it will not become a leader.
The network becomes nonfunctional.
Now, a process comes back online
Conflicting heartbeat and vote request...
Send heartbeat
Request vote
Pass leadership to higher term
Give Vote and become follower
Highest term always becomes leader

Fig. 5.5 Raft's Leader Election Process

Although this shows how the various logs can quickly become very inconsistent, these inconsistencies are resolved very easily. Essentially, the leader always keeps pushing all of its log entries to all followers. So when a follower comes back online again, its uncommitted log entries are replaced by the leader's log entries. Using the previous example of Figure 5.7, if all processes are online again and process C stays a leader, all processes will have the exact same Log as entry C.

Alternatively, one might contemplate what would happen if a process with a lot of uncommitted changes becomes the new leader, and tries to make changes to log entries that were already committed. In fact, this is not allowed by the protocol due to the following reason. A process only gives a candidate its vote, if its Term is higher or equal to its own Term. Therefore, because a majority is required *before* a value is committed, and one can only become a leader if it receives a majority of votes by processes with an equal or lower term, *no process can become a leader that does not have the last commit*. Because no process without a committed value can become a leader, and only leaders can write values to other processes, *no committed log entry can be overridden*.

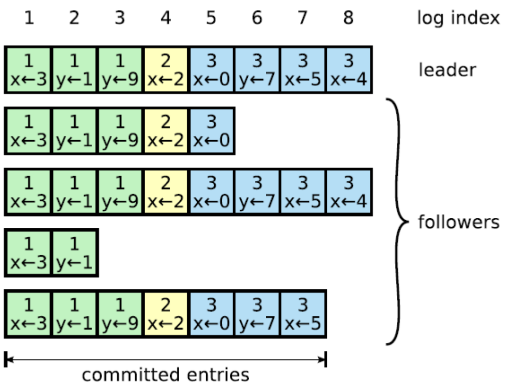


Fig. 5.6 Raft's Log [72]

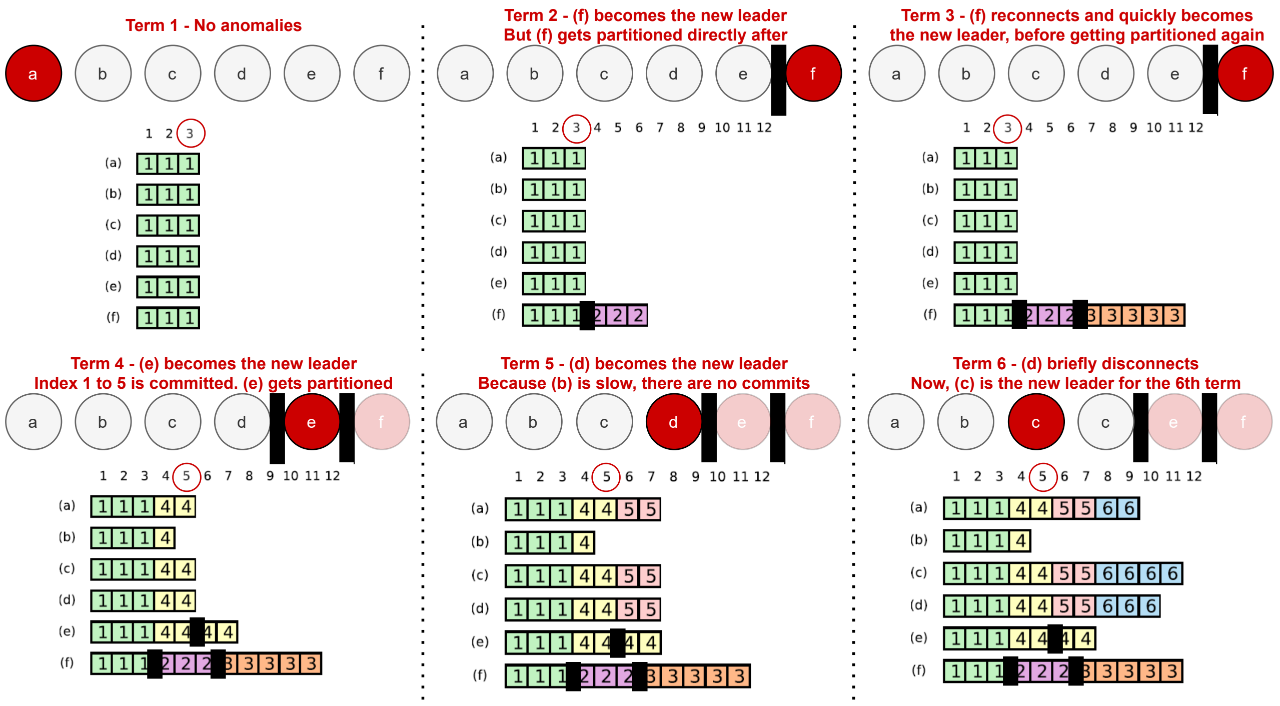

Term 1 - No anomalies
Term 2 - (f) becomes the new leader
But (f) gets partitioned directly after
Term 3 - (f) reconnects and quickly becomes
the new leader, before getting partitioned again
Term 4 - (e) becomes the new leader
Index 1 to 5 is committed. (e) gets partitioned
Term 5 - (d) becomes the new leader
Because (b) is slow, there are no commits
Term 6 - (d) briefly disconnects
Now, (c) is the new leader for the 6th term

Fig. 5.7 Example of forming inconsistent Raft Logs, inspired by [72]

5.1.5 Integration of Raft within Hyperledger Fabric

Now that Fabric's consensus protocol Raft has been explored in detail, we can look at how this protocol is used in Fabric. This consensus wrapper starts and maintains a unique etcd instance for every channel the OSN is defined in. For every channel, the OSN keeps track of the (local or remote) etcd instance.

As was seen in the transaction flow, an actor submits a transaction to a random OSN, rather than specifically to a leader. In Figure 4.6, we saw that the OSN forwards the transaction to the consensus-specific wrapper. As the consensus wrapper keeps track of the OSN with the leading etcd instance, the OSN can easily forward the transaction directly to the peer with the etcd leader. Note that this all happens outside of etcd so far. The reason for this is that the *transactions* themselves are not ordered by etcd, but *blocks* are, requiring some additional logic.

This could be confusing at first: If the goal of our consensus algorithm is to establish a total ordering of transactions, why complicate matters by ordering blocks, instead of the transactions themselves? To answer this question, we need to argue for the hypothetical design in which *transactions* would be ordered by Raft. Then, once a transaction would be committed by etcd, the transaction's ordering is final. Once committed, Raft calls the OSNs' State Machines, in this case, each OSN's the block cutter. The transaction is stored in this block cutter until the block needs to be cut. So far, so good. However, problems arise when we recall the conditions for cutting blocks. Either a block needs to be cut when it reaches its maximum size (which is not a problem), or a block should be cut once a timer exceeds since the first transaction entered the buffer. The latter condition was required to ensure a transaction is quickly added to a block, as opposed to waiting for a long time until the block is finally full (which could take hours if no new transactions are submitted).

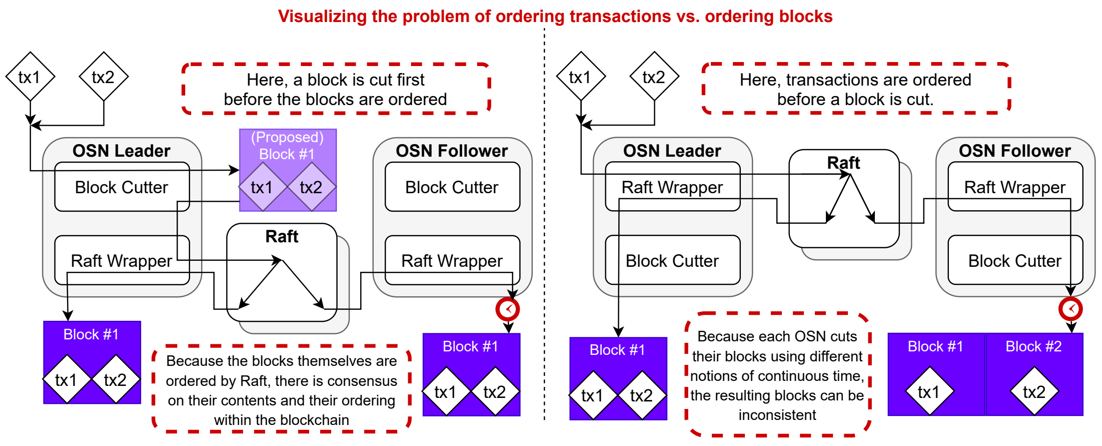

Visualizing the problem of ordering transactions vs. ordering blocks
tx1
tx2
Here, a block is cut first before the blocks are ordered
OSN Leader
Block Cutter
Raft Wrapper
(Proposed) Block #1
tx1
tx2
Raft
OSN Follower
Block Cutter
Raft Wrapper
Block #1
tx1
tx2
Because the blocks themselves are ordered by Raft, there is consensus on their contents and their ordering within the blockchain
Block #1
tx1
tx2
tx1
tx2
Here, transactions are ordered before a block is cut.
OSN Leader
Raft Wrapper
Block Cutter
Raft
OSN Follower
Raft Wrapper
Block Cutter
Block #1
tx1
tx2
Because each OSN cuts their blocks using different notions of continuous time, the resulting blocks can be inconsistent
Block #1
tx1
Block #2
tx2

Fig. 5.8 Problem of ordering transactions and creating blocks

In other words, blocks should be cut frequently to ensure transactions are quickly applied to the ledger, even when it is the only transaction in the block. Now, we have reached a dilemma: The OSNs have to cut blocks based on a synchronized timer, but Raft makes no continuous time guarantees! Because Raft cannot guarantee a change is applied to all nodes *consistently*, different OSNs might end up with transactions that are ordered differently over different blocks[6]. E.g. two transactions could be grouped in a single block on the first node using a fast connection, whereas two transactions might form two different blocks on the second node that has slow connectivity. This problem is captured by the right-hand side of Figure 5.8. Therefore, transactions must be included into blocks by the leading OSN, before they can be ordered by the Raft consensus protocol[7]. The left-hand side of Figure 5.8 shows how blocks are actually created in Fabric.

5.1.6 Security Implications of Raft for Fabric

At this point, it should be clear what Raft is, and how it is used within Fabric. This also reveals why Raft is CFT and not BFT. Namely, if only one dishonest node is part of the consensus cluster, the entire Raft cluster might be undermined. Although a formal analysis of attacks on the Raft consensus protocol is too much out of scope for this thesis, some drawbacks immediately become apparent, as discussed next.

As a first drawback, an adversarial process is able to take leadership from another node and thus control all blocks (and their transactions) added to a channel. Recall that once a process does not receive a leader's heartbeats, it becomes a candidate, starts a new election and increments the term. Only a *single* process is required to initiate this procedure! Combine this with the fact that a leader steps down and becomes a follower if a candidate has a higher Term. Thus, if an adversary is not a leader, it can artificially increase its term and start an election to become the new leader. Thus, it is easy for an adversary to take over control over a channel's ordering service. Because the adversary is able to push writes to all followers, it could abuse its leadership position to muddle the log entries with spammed (i.e., invalid) data.

[6]Hence, this shows the result of dropping Consistency of the CAP-theorem.

[7]Perhaps a second raft cluster could be used per channel to reach consensus on the ordering and contents of blocks, which would make ordering transactions possible but this (1) becomes too complicated and (2) defeats the purpose of ordering transactions in the first place.

A slight variation on the previous drawback, is that any adversarial process that is part of the ordering service, can perform a Denial of Service attack. Since all write operations may only be performed by the leader, rejecting all transactions from clients means that no new blocks are appended to a channel by the leader of that Term. Combine this with taking ownership once another process is the leader, and this Denial of Service attack may halt the blockchain for an indefinite amount of (consecutive) terms. A slightly different version of a Denial of Service attack could be to continuously increase the term and start a new election, e.g. every 50ms. Then, no matter who becomes a new leader of a term, all processes are constantly in the election phase, resulting in nobody accepting write requests.

Although it might seem like this makes the ordering service a very weak link in the transaction flow, this does not *necessarily* have to be the case. Since *committed* log entries may never be overwritten, the blockchain remains valid up to the point when adversarial actions start (valid and committed blocks are not removed from honest processes). It simply means that honest organizations operating the ordering service, in the worst case, write useless information, omit new transactions (of competitors or even everyone), or bring the blockchain to a halt. Thus, the damage that an adversary can perform is limited. Secondly, it means that the ordering service should not be operated by any organization with conflicting interests: The OSN should not have a (financial) interest in filtering the transactions and blocks that are, or are not, written to a channel. Consequently, this brings us back to the discussion of requiring trust in *every single* participant of the ordering service. Additionally, this discussion highlights why Raft could be a *good enough* solution for permissioned networks, but also why the Raft algorithm used by Fabric, as proposed by Ongaro et al., would not work at all in permissionless networks.

5.2 Reaching consensus in Besu

Besu supports no less than *four* different consensus protocols. These include Proof of Work, Clique, IBFT and IBFT 2. Since sifting through all of these consensus protocols individually would be too much, a bird's eye view of the protocols is given instead. As it turns out, each of the four protocols is inspired by its predecessor. Clique was created because Proof of Work caused issues in Ethereum's test networks. IBFT was created to adapt Clique for enterprise requirements. And IBFT 2 is an adaptation of the original IBFT to eliminate some of its limitations. Because each protocol succeeds its previous protocol, we introduce them in chronological order as well.

5.2.1 Introduction to Proof of Work for public networks

When Bitcoin's paper was introduced back in 2008, Nakamoto introduced Proof of Work (PoW) to *"agree on a single history of the order in which (transactions) were received"* [68]. By ordering transactions into blocks and including the previous block's hash, a total order is established. However, there could be two blocks that include the previous block's hash, thus, agreement is required on which block(chain) would be considered correct. The original solution to this problem was that as long as the majority of the network's nodes are honest, their chain would always be longer than that of a minority group of adversaries.

The next challenge was to determine how a majority would be measured. Nakamoto initially considered a voting process in which one IP-address may cast one vote at each block height, but saw it could be easily undermined by allocating many IP addresses [68]. Instead, a majority should be defined in such a way that makes it (financially) infeasible for adversaries to gather more weight than all honest nodes combined. This resulted in the notion that a "vote" should not be made per person, but per CPU. Essentially, it means that an adversary needs to gather more computational power than all honest nodes combined, in order to effectively fork the blockchain and make it outgrow the honest chain. The computational puzzle that miners need to solve is to find a nonce that when hashed, it starts with a *difficulty*-amount of zeros. This notion of agreeing on the longest chain by making block creation difficult through computational power is referred to as PoW.

Because PoW had been effective for multiple years, the developers of Ethereum built upon this approach to create their own PoW-implementation *Ethash* [97]. Because the Ethereum developers wished to test features before rolling them out on the main network, they created test networks. In 2017, test network *Ropsten* was started using the same PoW-algorithm as the MainNet. However, it turned out that the PoW consensus protocol that was a great success on the main net, caused for many issues on the Ropsten test network [89]. For one, it has faced many 51% attacks that caused an older chain to take over an honest chain. Secondly, adversaries increased a block's gas limit to be about 2000 times larger than the MainNet's gas limit, causing excessively large transactions to slow the test network.

To understand why PoW failed multiple times on Ropsten, we need to look at it from a financial perspective. Recall that PoW was designed to make it financially infeasible for an adversary to obtain a majority "vote". As a result, the process of creating blocks requires significant computational power, which costs a lot of energy. Over time, this energy becomes expensive. The MainNet makes the process of block mining more attractive by awarding the block creator with a reward, i.e. a small amount of the underlying cryptocurrency. Because this underlying cryptocurrency has no value whatsoever on test networks[8], there is no incentive for the general public to dedicate their expensive hardware and energy. As a result, Ropsten's *Hashrate* is significantly lower than the MainNet's, making it very easy for an adversary to perform various attacks. As written by Szilagyi, "*Proof of Work cannot work securely in a network with no value*" [89].

Although one could argue a permissioned network has inherent value by its very nature while neglecting the cryptocurrency, the same argument could be made for permissioned networks. For PoW to have significant value, participating organizations need to waste a lot of energy to make PoW meaningful, such that no other competitor can overtake the network. This is especially true if we consider that organizations are likely to be of different sizes with different magnitudes of resources at their disposal. Besides, collaborating participants of permissioned networks are identifiable by definition. The idea that organizations are creating a *collaborative* network, resulting in a fierce *competition* for spending the most hashpower becomes a rather silly paradox. Therefore, PoW will not further be considered for *permissioned* networks. Still, it is worth mentioning that enterprises are able to use Besu and its private transactions (as explained later) on Ethereum's *permissionless* MainNet using PoW, if they wish to do so.

5.2.2 Clique as Proof of Authority for Ethereum's Test Networks

To solve Ropsten's issues caused by PoW, a different consensus protocol was required. *Clique* was designed to fill this gap for Ethereum's test networks. Instead of relying on a proof of *work*, Clique takes a different approach by requiring a proof of *Authority*. In practice, blocks are no longer created through *mining* for nonces by *miners*, but are created by *signers* that are simply *signing* blocks (another word for signing is *sealing*). This means that block creators no longer have to waste energy by finding a hash that matches the required difficulty.

Essential to Proof of Authority (PoA) is that a block may *only* be created by *trusted* signers, rather than just anyone that is a member of the network. Since Clique was originally designed for a permissionless network, the decision making whether a node was considered *trusted* or *untrusted* had to be performed reliably in a decentralized fashion. To that end, a list of trusted signers is incorporated into the blockchain itself to ensure the entire network shares the same view on this list. The main role of the Clique consensus protocol is (1) to define the rules on how to maintain the signers list and (2) to restrict the validity of a block by requiring a signature of a member of the signer's list.

The list of trusted signers should be modifiable, meaning it should be possible to add a new node to the list. In addition, a node's trusted status should be revocable, e.g. when a node (or its private key) is compromised. To facilitate these changes, Clique introduces a voting process in which only trusted signers may cast votes. To cast a vote, a node mentions the address of another node and whether it votes for the node to be trusted or not. At some point, all ballots should be final and counted to make the final decision. In Clique, these voting rounds are called *Epochs*. By default, an epoch is set to 30.000 blocks. During each epoch, only the block creator may cast exactly 1 vote due to technical limitations, explained later. After an epoch has expired, the votes are counted, the list of trusted signers is updated, and a new voting round starts. To prevent votes with only a small number of signers, or someone voting on a new address just before the end of the epoch (so it only

[8]One can request free Ether by entering his or her EOA's address in a number of faucets.

has 1 total vote), the result of a vote is only effectuated when there is consensus on a majority of trusted signers, i.e. a majority of trusted signers backs the final decision. An example voting timeline is provided by Figure 5.9.

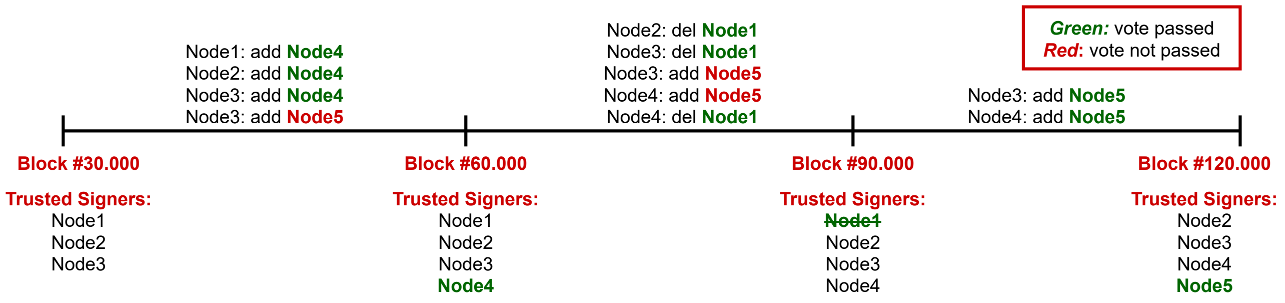


Fig. 5.9 Example timeline of Voting Epochs in Clique

The block creation itself works as follows. Every signer maintains the complete list of trusted signers, as well as the current block number. Every block number maps to one signer by taking `blockNumber % trustedSignersSize`. This is called the *in-turn signer*. If the in-turn signer does not create a block for some amount of time, any other *out-of-turn signer* may create the block instead. When the in-turn and an out-of-turn signer create a block simultaneously, the in-turn signer's block is always preferred. To prevent all out-of-turn signers creating a block simultaneously, each signer waits a random amount of time (`rand(SIGNER_COUNT * 500ms)`) [89].

According to Angelis et al., a majority of clique nodes need to be Byzantine in order to take over the network [27]. Instead of focusing on proving this aspect, we mention how Clique prevents two types of attacks, as mentioned in the EIP [89]. To prevent a compromised node from creating a lot of blocks (to have the longest chain), each trusted signer may sign only one out of `floor(SIGNER_COUNT / 2) + 1` blocks. This ensures that a majority trusted signers have created a block, before the same trusted signer can create another one. Another attack could be to censor or ignore votes that are unwanted. By recording votes in blocks and only allowing a node to create a block once in a while, a majority of nodes must be Byzantine to censor out these votes.

Implementation-wise, an interesting observation of the voting process is that all voting is done through unused PoW-header fields of blocks. Because Clique was designed for a permissionless network that could quickly grow very large in terms of blocks, light and fast syncing had to be supported [89]. Using these sync strategies, only block *headers* are downloaded and validated. Therefore, the votes could not be recorded in a smart contract on the chain itself (since light/fast sync do not have access to smart contract state), but had to be recorded inside header fields. Then, a requirement of the consensus protocol was that it should be easy to integrate with existing Ethereum client implementations. Adding new fields to the block header would change the RLP-structure of blocks, which would be bad maintainability-wise. Thus, the EIP was approved by the Ethereum community once all voting information was included in existing PoW-header fields. A direct consequence of this is that for each block, only the block creator can cast a vote. Thus, a node that wishes to vote has to wait for its turn to create a block, before casting the ballot.

5.2.3 Istanbul Byzantine Fault Tolerant for Enterprises

In 2017, Clique was deployed on the Ropsten test network with success. The concept of PoA in which block creators no longer have to search for a nonce, but can simply sign a block, was attractive for private Ethereum networks. Because block creation was no longer a difficult and time-intensive task, blocks could be created faster to achieve a higher throughput [24]. Yet, it lacked one attribute important for enterprise settings: immediate block finality. Enterprises typically want their transactions to be final, and not have them undone at some point in the future. Two months later, Clique was adapted to ensure a block is immediately final after commit. The resulting consensus protocol is Istanbul Byzantine Fault Tolerant (IBFT).

IBFT reuses the same voting mechanism of Clique for maintaining a list of trusted signers. Yet, the tasks of the trusted signers have changed. It is no longer enough for a block to be created by any of these signers: IBFT also requires a majority of nodes to agree on the block, i.e. validate it, before the block is considered final. This validation scheme of blocks is inspired by a 1999 paper on Practical Byzantine Fault Tolerance by Castro et al. [20]. Time is divided up into *Rounds*. During each round, the nodes try to come to an agreement on a proposed block by performing Castro et al.'s three-phase consensus algorithm. The three-phase consensus algorithm consists of a *pre-prepare* phase, a *prepare* phase and a *commit* phase. The conditions for transitioning between these phases, as well as the communication of each phase, is explained below and illustrated in Figure 5.10. At the end of the chapter, a more detailed (but also slightly more complicated) view on the transitions is given.

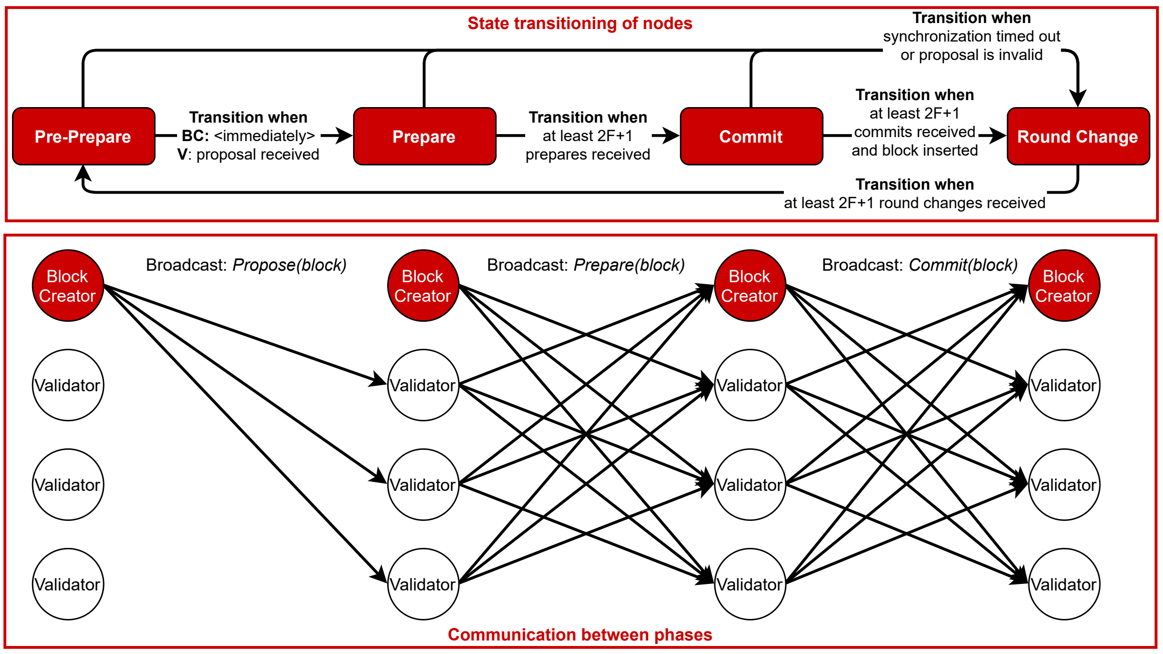

State transitioning of nodes
Transition when synchronization timed out or proposal is invalid
Pre-Prepare
Transition when BC: <immediately> V: proposal received
Prepare
Transition when at least 2F+1 prepares received
Commit
Transition when at least 2F+1 commits received and block inserted
Round Change
Transition when at least 2F+1 round changes received
Block Creator
Validator
Broadcast: Propose(block)
Broadcast: Prepare(block)
Broadcast: Commit(block)
Communication between phases

Fig. 5.10 Communication and transitions within IBFT

Three-phase commit

During the *Pre-prepare* phase, each node runs a (local) deterministic function to decide who is the next block creator. The block creator then creates a block and broadcasts this proposal to the rest of the network. All non-creators, i.e. validator, are waiting to receive a block during this phase. If a validator however does not receive a proposal for some amount of time, it could mean that the block creator is offline or the message got lost. In this case, it broadcasts a *Round Change* message.

A node immediately enters the *Prepare* phase once it has broadcast the proposal (in case of the block creator), or received a proposal (in case of the validator). During this phase, all validators verify the proposal. In case the proposal was *valid*, it submits a *Prepare* message. If the validator found the proposal to be *invalid*, it submits a *Round Change* message. Because the block creator already knows it considers the block as valid (since it created the block itself), the block creator submits the prepare message without performing further validations.

During the *Commit* phase, all nodes are waiting to receive *at least* `2F+1` Prepare messages. After receiving enough Prepare messages, the node knows that enough other nodes agree that the block is valid and are willing to (try to) commit the block. Therefore, the node locks the block, meaning that it will not consider another block (unless insertion fails). To let the rest of the network know it received enough prepare messages (and is thus not partitioned), and that it has locked the block, it sends out a *commit* message. After receiving enough (`2F+1`) commit messages, the block is appended to the ledger and is considered final. When this commit phase was successful, a new round is started immediately after a timer (for throttling the block creation) expires. If the block insertion was unsuccessful or if waiting on commit messages took too long, the block is unlocked and the node votes for a round change.

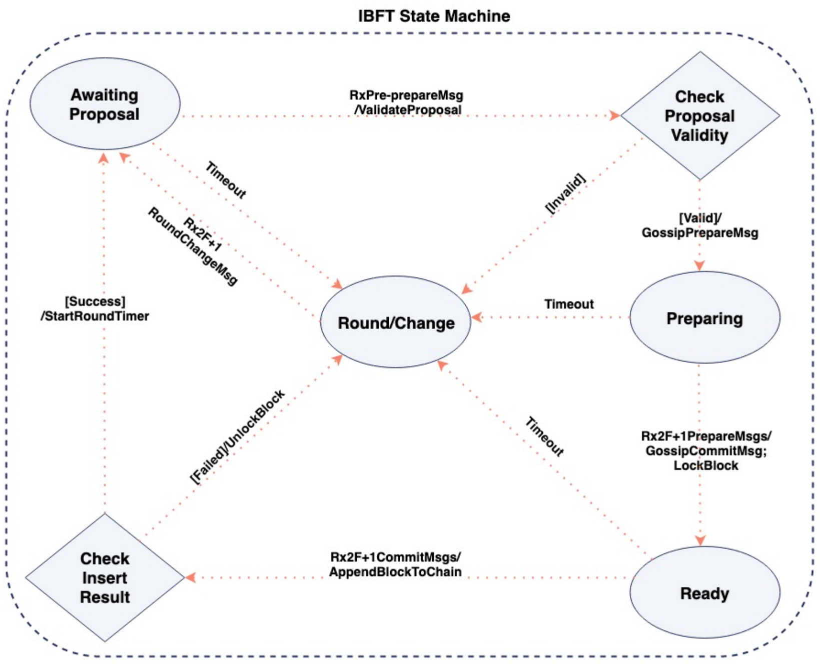


Fig. 5.11 Another view on the possible transitions of a node by Ledgerium [33]. Awating Proposal maps to the Pre-prepare phase, and Ready to the Commit phase.

As was explained for earlier phases, a round change may be triggered before a round is complete. These are, however, rainy day scenarios. As such, a node does not immediately abort a round once it receives a RoundChange message from another node. Instead, if a node *can* successfully continue without issues (not receiving a proposal, timing out), it will continue to do so. Only after a node has received `2F+1` RoundChange messages during the same round, it concedes and agrees to move on to a second attempt in a new round. To make this more concrete, consider the case in which some node `N1` triggers a round change early, because it does not receive the block creator's proposal. At the same time, all other nodes including node `N2` did receive the leader's proposal and are continuing the happy path. Because IBFT allows up to `F` faults, healthy node `N2` requires at least `2F+1` round change messages before aborting the current round. Consequently, a malicious adversary cannot bring the network to a halt by continuously changing rounds. This shows the BFT nature of IBFT. Figure 5.11 provides the reader with another view on the three-phase commit by showing every single transition, now that the entire process has been summarized. Because this three-phase commit eliminates illegitimate block (proposals) early on, it is no longer required to wait for `N/2` other nodes to create a block, before a node may create a block again. Instead, this is relaxed to dictate the same node may not create 2 blocks in a row.

Issues of IBFT

Note that so far, we mentioned IBFT inherited Clique's voting scheme which satisfies the BFT property. On top of the voting system, IBFT aims to provide BFT consensus on immediate block finality. However, despite what the name IBFT and various sources suggested at the time [89, 24], it turned out that IBFT was not BFT after all. Saltini et al. identified two main issues with the original IBFT algorithm related to persistence and liveness [81].

First, let's discuss the issue with persistence. Saltini et al. define persistence as follows: If an honest node adds a transaction at some position in the blockchain, then (1) it is the only transaction it may ever add at that position and (2) all other honest nodes will eventually have that transaction in the same position as well. Thus, this property essentially describes finality of transaction ordering. In order to have `t-BFT` persistence, the persistence property will hold if no more than `t` nodes are Byzantine. To briefly summarize Saltini et al.'s proof, a block is *committed* (finalized) once *majority* is reached. Yet, in some edge cases, a Byzantine node can approve two different blocks at the same height, when all other honest validators are split in two sets. Two scenarios in which this can occur is (1) when a block is simultaneously created and prepared in an evenly split network due to eventual consistency, or (2) if the Byzantine node is the block creator and deliberately proposes two different blocks and sends the first block to the half of the other nodes, and the second block to the other half. Thus, if the Byzantine node has to give the decisive choice between an evenly split network, it can approve two different blocks (twice), causing two sets of honest validators to commit different blocks. Because two groups have then committed different blocks, each group consider their blocks final and, from that moment, do not communicate anymore to the other group because their root state hashes differ. Figure 5.12 by ConsenSys summarizes this concept very well. This contradicts the definition of persistence that there can only be 1 transaction at some position in the blockchain across all honest nodes.

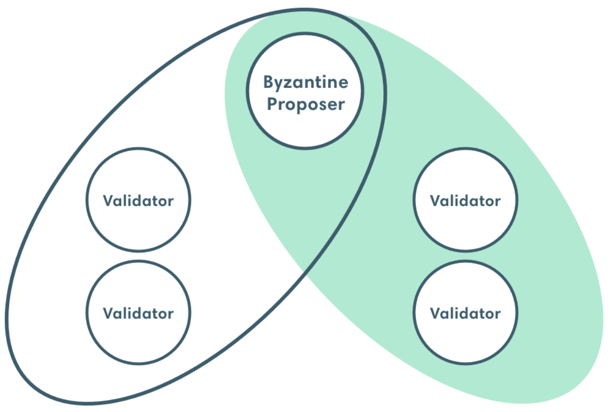


Fig. 5.12 A single Byzantine node may break BFT-consensus on blocks using the IBFT protocol [25]

The second issue relates to the liveness of the blockchain. Saltini et al. define liveness as the attribute in which it should always be possible to add a newly submitted transaction to the ledger, eventually. In order to satisfy `t-BFT` liveness, the liveness property should hold true when there are less or equal than `t` Byzantine nodes. The proof by contradiction considers a setting in which the nodes are grouped as a majority `Mj` including 1 faulty node (not even Byzantine!) and a minority `Mn` of nodes. Now, assume all nodes send the Pre-prepare message (i.e. the block is valid), but only `Mn` receive a majority of these messages before the timeout. Then, all nodes in `Mn` lock the ledger on that block, while all nodes in `Mj` start a new round, propose a new block and lock that block in. When the faulty node crashes before sending the Commit message, it means both groups of validators are at the stage in which they have locked in a block, but neither group can reach a majority. As a result, no new blocks and transactions may be appended to the ledger.

Both issues described above may look similar, but are very different. The first results in a blockchain splitting in two forks, each continuing to create blocks, as long as a Byzantine node keeps voting and approving for both groups. The second is an unlucky accumulation of events resulting in a standstill of the ledger. Either way, in both cases the added voting mechanism on top of Clique fails to fulfill its purpose. It aims to guarantee consensus on a single chain without forks in the presence of Byzantine nodes. Only one Byzantine node is required to undermine this, and only one faulty (or Byzantine) node is required to halt operations. More importantly, PegaSys and Clearmatics frequently encountered issues like this while testing the protocol, highlighting these issues do not merely exist on the theoretical level [25]. Therefore, (this version of) IBFT should not be used and will not be considered further in the remainder of this thesis.

5.2.4 Istanbul Byzantine Fault Tolerance 2 to improve its Predecessor

Despite the failure of IBFT 1.0, reaching immediate block finality was still a requirement for which there was no (viable) solution. Saltini et al., who also proved IBFT 1.0's flaws, set out to improve the protocol by overcoming the aforementioned limitations, in terms of BFT persistence and BFT liveness [82]. The resulting protocol was named IBFT 2.0. As IBFT 2.0 is an improvement or update to the 1.0-version of the protocol, most concepts remain unchanged, such as the three-phase commit and the various transitions. Therefore, this section does not re-introduce IBFT as a new algorithm, but rather the most important changes made to version 1.0. The reader is encouraged to have read Section 5.2.3 before going through this section.

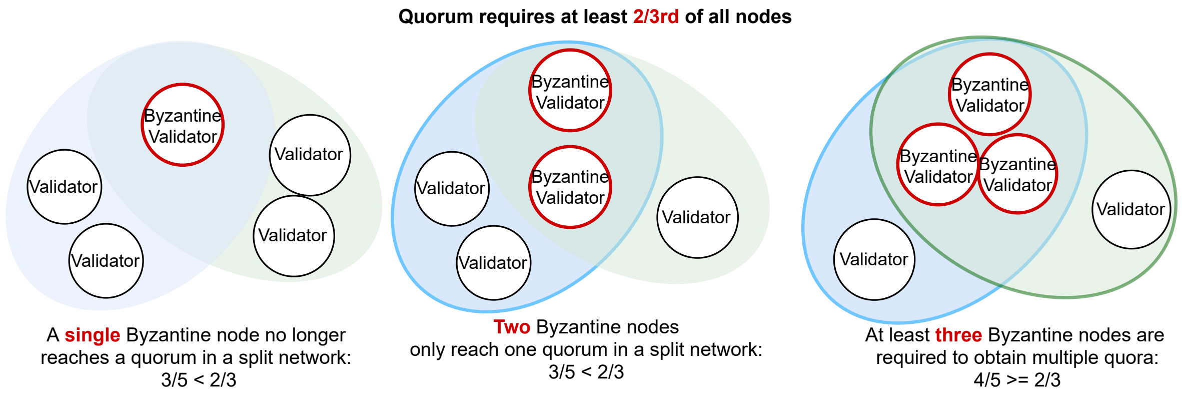

Quorum requires at least 2/3rd of all nodes
Byzantine Validator
Validator
Validator
Validator
Validator
A single Byzantine node no longer reaches a quorum in a split network: 3/5 < 2/3
Byzantine Validator
Byzantine Validator
Validator
Validator
Validator
Two Byzantine nodes only reach one quorum in a split network: 3/5 < 2/3
Byzantine Validator
Byzantine Validator
Byzantine Validator
Validator
Validator
At least three Byzantine nodes are required to obtain multiple quora: 4/5 >= 2/3

Fig. 5.13 Reaching multiple quora on IBFT 2.0

First, let's see how the persistence issue is resolved in version 2.0. In IBFT 2.0, it is not longer satisfactory for a *majority* to satisfy a quorum, but `CEIL(2n/3)` nodes are required, i.e. *at least* 2/3rds. This makes it much harder for Byzantine validators to reach the new quorum, especially as the number of nodes in the network grow. This is illustrated by the following examples. If we have a network of 5 nodes, we must have `CEIL(2*5/3)=4` to reach the quorum. This means that it is no longer possible for a single Byzantine node to reach two quora and vote for two chains: at least two Byzantine nodes are required. Figure 5.13 also shows that two Byzantine nodes cannot reach two quora on two disjoint sets of honest validators. Thus, it can still not perform the persistence and liveness attacks mentioned beforehand. The third part of Figure 5.13 shows that at least 3 nodes are required in order to establish a quorum in two disjoint honest validator sets. Only in the third image, Byzantine nodes can split the blockchain in two or bring the blockchain to a halt using the attacks mentioned in Section 5.2.3. Therefore, a network of 5 nodes is said to have the *2-BFT persistence* attribute, meaning it still guarantees the persistence property even if there are two Byzantine nodes.

Recall that for the liveness issue, nodes did not necessarily have to be Byzantine. The blockchain could be brought to a halt due to the failure of a single node. To make the protocol guarantee *t-BFT liveness*, a similar argument is made involving faulty nodes, rather than Byzantine nodes. Namely, changing the quorum cannot result in two sets of honest peers locking in a block due to a *single* faulty node, regardless of the network size. Rather, the amount of faulty nodes required to result in this gridlocked state increases as the network grows. Although the protocol now supports *t-BFT liveness* based on the network size, Saltini et al. introduced another optimization to the three phase algorithm. Recall that blocks had to be locked by honest nodes in version 1.0 to ensure that once a node committed a block, the decision on the block is final (unless insertion fails locally). An optimization is that once a block is considered *committed* by at least one node (2/3rds have sent a prepare message), and a new round starts, the block agreed upon in the new round will always include the committed block (of course, only up to `t` Byzantine nodes). This allows the entire locking mechanism to be completely removed.

Since IBFT 2.0 solves the issues of version 1.0 related to persistence and liveness, it could be an interesting improvement over Clique, especially for permissioned networks. Namely, after a transaction has been recorded on the blockchain, the submitter can rest assured that it will stay there and may not be reverted due to a soft fork in which another chain became the longest chain. In addition, the IBFT 2.0 consensus protocol is backed by mathematical proofs to guarantee BFT.

5.3 Findings

In this chapter, we have seen three different consensus protocols viable for permissioned blockchain networks. For Fabric, Raft is the only recommended consensus protocol. In Besu, four consensus protocols are provided out of the box: PoW, Clique, IBFT 1.0 and IBFT 2.0. We have seen that PoW is highly inefficient in a network without a cryptocurrency, and IBFT 1.0 is inherently flawed in terms of coming to a halt and Byzantine actions. Both of Besu's Clique and IBFT 2.0 protocols are considered since they provide an interest contrast in the overhead of Besu operations, with the former being much more lightweight than the latter. Therefore, the remainder of this chapter focuses on only comparing Raft, Clique and IBFT 2.0.

5.3.1 Integration of consensus protocol with the blockchain framework

Algorithm Implementation

Starting at the highest level of abstraction, we can look at how each consensus protocol is integrated with and used within their blockchain frameworks. Fabric has historically preferred to use a consensus protocol of an external software vendor as we have seen with Kafka and Raft. Originally, a Kafka cluster had to be deployed and managed completely independent of the Fabric cluster. For Raft, Fabric uses an implementation provided by etcd. Although this is an implementation of an external vendor as well, management and orchestration of the Raft cluster is performed automatically by an OSN. Besu on the other hand does not use any external software components for implementing its consensus algorithm. Consequently, Fabric's consensus protocol(s) are backed and maintained by a larger community, whereas Besu's protocols could be more easily tailored for its individual use case inside the blockchain framework.

Consensus Protocol Type

All three consensus protocols rely on fundamentally different concepts. Raft takes a leader-follower approach. The leader corresponds to the block creator in Fabric, making it easy for a single node to establish an ordering that should be copied by all other followers. At Clique's core, there is a simple deterministic function to decide who is allowed to create the next block. As such, nodes can execute this function locally without any communication (besides the distribution of blocks, which is not inside the scope of the consensus algorithm). IBFT 2.0 reuses a three-phase commit algorithm, as proposed under Practical Byzantine Fault Tolerance in 1999 [27]. Combined, these phases result in a single decision by the network as a whole.

Data to consent on

As the consensus protocols rely on different foundations, they aim to reach consensus on different types of information. Raft aims to establish a total agreed-upon ordering of blocks. Clique establishes consensus on two pieces of information: the list of trusted signers and the correct chain of blocks (i.e. longest). IBFT 2.0 also knows two-fold consensus, namely a list of trusted validators and the validity of blocks (its ordering is implied: a valid block means its order is established).

Trigger of Block Creation

As we have seen in Chapter 4, the consensus protocols of both frameworks are actually triggering the block creation. Thus, we could also look at the conditions of each consensus protocol that results in the invocation of the block creation. In Fabric, a block is created, or rather cut, as soon as it has reached a maximum size in terms of bytes or number of transactions, or if a transaction was sitting in the buffer for at least some amount of (continuous) time. For both Clique and IBFT 2.0, a fixed delay in seconds is configured in-between blocks. When such a timer has expired, a new block is created. An interesting observation is that Fabric's block creation depends on the input load: the more transactions a leader receives, the more blocks it (aims to) create. In Besu, this is not the case at all. If there is a high load, or even if there are no transactions waiting at all, blocks are created at approximately the same rate.

Discrete Division of Time

A common difficulty of *eventually synchronous* distributed systems relates to how systems determine when a period of (discrete or continuous) time has elapsed and when a new period should start. Relying on synchronizing system clocks is often unreliable. Therefore, all consensus protocols divide time up into discrete time periods that are unrelated to our notion of continuous time (i.e. seconds). In Raft, the goal is to divide time into discrete *Terms*. These terms essentially map to the n-th election and allow nodes that were unavailable for long periods of time to understand their location within the timeline, i.e. how far their log is behind. In Besu, time needs to be divided into discrete *epochs* to accommodate voting rounds. Epochs allow nodes to give a limited time frame for voting on the trusted signers after which all ballots are cleared. Similarly, IBFT 2.0 has uses the same epochs for voting on trusted validators. In addition, time is discretized into *Rounds* so that validators know when it is lagging behind in an older three-phase commit flow. All in all, the same pattern of time discretization reoccurs across all three consensus protocols, but obviously for different reasons.

Overview of differences on consensus protocol integration

Consensus Protocol Integration			
Concept or Feature	**Fabric/Raft**	**Besu/Clique**	**Besu/IBFT 2.0**
Algorithm Implementation	External vendor, but orchestrated and managed by Fabric.	Embedded algorithm.	
CP Type	Leader-Follower.	Simple deterministic function.	Three-phase commit.
Data to Consent on	Atomic block ordering.	List of trusted signers, longest chain of blocks.	List of trusted validators, validity of blocks.
Trigger of Block Creation	Byte size reached, transaction count reached, or delay of oldest transaction exceeded.	Timer (back-off period) in-between blocks expires.	
Division of Time	Discrete terms for elections.	Discrete epochs for voting.	Discrete epochs for voting and discrete rounds for committing.

Table 5.1 Overview of differences on consensus protocol integration

5.3.2 Adaptability of consensus protocols

Consensus Protocol Elasticity

An important requirement for blockchain networks, and thus for the consensus protocol, is high elasticity, i.e. the ability to (dynamically) add or remove nodes. In both protocols of Besu, there are no bounds on this. During a single voting round, any number of validators may be removed or added to the trusted list, provided a majority vote for their addition or removal. In Fabric, this is slightly more complex. Essentially, only a single Raft node may be added or removed at a time[9]. This is caused by the fact that existing Raft nodes need to reach consensus each time a change is made to the new list. Although this seems like a small restriction, there is a larger risk associated with it. If an update to the amount of nodes does not reach consensus on a proposed change, the Raft cluster becomes nonfunctional. To name an example: no new nodes can be added to a cluster if 2 out of 3 nodes are offline. Or a second example could be when the cluster has a size of 2: if one *could* add two nodes simultaneously, the new cluster size would be 4. If they weren't added correctly (their certificates are wrong), the cluster can never reach a majority again and becomes nonfunctional as well.

[9]https://hyperledger-fabric.readthedocs.io/en/release-2.2/raft_configuration.html#reconfiguration

Modifications to the consensus protocol implementation

Making modifications to a consensus protocol could be an important requirement for enterprises that wish their network lasts for years to come. When requirements change, or optimizations have been discovered, a business might want to apply these to existing networks. This includes creating a (new version of) a consensus protocol, as well as changing the protocol while a network is live. First, let's consider modifying or creating a new consensus protocol for the framework. This process is similar to both frameworks: the changes namely have to be made in the source code of the framework itself. Both frameworks do not allow an external hook or plug-in to be added without modifying the source code. Second, an existing network would have to allow the consensus protocol to be changed for another implementation. In Fabric, this was historically possible as the deprecated Solo and Kafka protocols could be changed for Raft[10]. Yet, the documentation explicitly mentions *"[Switching consensus algorithms] is explicitly not supported."*[11]. In Besu, no information is provided on this aspect. However, because information specific to a consensus protocol is embedded in the header of each block, swapping to a different implementation would be difficult. Theoretically, the old protocol could be used (for syncing clients) up to a predefined block, at which another protocol would be enabled (similarly to how the Ethereum protocol is upgraded), although this would definitely be a nontrivial task[12].

Overview of Consensus Protocol Adaptability

Consensus Protocol Adaptability			
Concept or Feature	**Fabric/Raft**	**Besu/Clique**	**Besu/IBFT 2.0**
Cluster Elasticity	Only 1 node removed or added at a time.	Unbounded, but change delayed until end of voting epoch.	
Modifying or Creating CP	Consensus Protocols cannot be developed (and deployed) independently of the framework's source code (as a plug-in), but requires the blockchain framework's source code to be forked and changed.		
Swapping CP of Running Network	Historically possible, but not guaranteed or supported for the future.	Unsupported and undocumented, although technically possible through hard fork.	

Table 5.2 Overview of Consensus Protocol Adaptability

5.3.3 Functionality of the consensus protocol

Ordering and Validity of blocks

Next, we can look at different functionalities that consensus protocols (must) perform for their framework. First, all protocols must support *ordering* of (transactions and) blocks. The reader should be aware what this entails, as the term "ordering" could be a bit ambiguous. This does *not* mean that the consensus protocol receives a number of blocks and it needs to decide how to order/shuffle these. None of the consensus protocols do this. With ordering, the goal is to *establish a total order of blocks*, as each block links its parent block's hash. In Raft, for instance, a leader may have created multiple (uncommitted) blocks that will be overridden and never committed. Thus, the reader should not mistake this for taking a number of blocks and putting them in any (random) order. Ordering is about the *content* of the block: what block will be *ordered* next in the chain? Raft and Clique both do not aim to reach consensus on the block's contents, i.e. all transactions in the created block are not *guaranteed* to be *valid*. Fabric would discard such a block on each peer after distribution, whereas in Clique it would be a soft fork between nodes that recognize the block as valid (e.g. creator) and those that do not. IBFT guarantees all committed blocks are valid. As a result, a Fabric network that is flooded with invalid transactions

[10] https://hyperledger-fabric.readthedocs.io/en/release-2.2/kafka_raft_migration.html
[11] https://hyperledger-fabric.readthedocs.io/en/release-2.2/Fabric-FAQ.html#ordering-service
[12] The Ethereum community plans to swap Proof of Work out for Proof of Stake on their running MainNet. Similar steps could be taken to introduce a different consensus protocol in permissioned networks

(e.g. spam) would record all of them on the ledger (although flagged), in Clique only the spammer stores and executed the invalid transactions, and in IBFT 2.0 those invalid transactions are dropped immediately and do not leave a trace.

Block Creation

Other functionality and responsibility of all consensus protocols discussed relate to deciding on who is the block creator. Block creators in Raft are elected. This entails that any node may become a candidate. Once this candidate has received a sufficient amount of votes, it will promote *itself* as block creator. In Clique and IBFT 2.0, block creators are *assigned* using a Round-robin fashion of taking turns. Every new round, the next trusted validator is given preference over an out-of-turn validator. In Clique, checks are in place that require half of the trusted signers to have signed a block, after any node has signed one. In IBFT 2.0, this restriction is relaxed such that no node may create two blocks in a row, thanks to the three-phase commit. The primary goal of Besu's Clique and IBFT 2.0 throttling is to prevent a single (Byzantine) node from having a too large impact on the network such as a monopoly-like position, e.g. to prevent a Byzantine node from discarding all transactions from a competitor.

Replication

We saw that Raft is not only an algorithm for reaching agreement on a single decision, but to implement an entire RSM. As such, Raft is also responsible for replicating the (past) decisions (or blocks) to all other nodes of the consensus cluster[13]. Both protocols in Besu only replicate (or share) the decision itself, i.e. new block. Older blocks may be requested and shared as well, but is not the consensus protocol's responsibility.

Communication with other nodes

We already hinted before that Clique has close to no communication overhead, because the voting system is embedded in the block distribution (not part of the consensus protocol) and a local deterministic function. For the other consensus protocols, this is not the case. As a Raft leader continuously distributes the log to all followers, it does have a communication overhead as Log replication is part of the consensus protocol. Yet, when a node remains the leader, sharing this block only happens once per follower. Once network issues occur, however, there could be lots of additional voting rounds and overriding of log entries. More importantly, a Raft leader is directly connected with all followers. In IBFT 2.0, messages of the consensus protocol are only passed to a subset of other nodes using a gossip protocol, to prevent any node from having to send `N` outgoing messages at each phase. Due to requiring three (largely synchronized) phases of communication, there is a significant message overhead in IBFT 2.0.

[13]Initially, only a limited amount of nodes (like 20) are immediately available to share. If another Raft node requires older blocks, it downloads a snapshot of the Log.

Overview of Consensus Protocol Functionality

Consensus Protocol Functionality			
Concept or Feature	**Fabric/Raft**	**Besu/Clique**	**Besu/IBFT 2.0**
Consensus on Ordering of blocks	Yes.	Yes.	
Consensus on Validity of blocks	No.	No.	Yes.
Deciding on Block Creator	Election. Any node may ask another node for its vote to become the block creator.	Assignment. Round-robin fashion of taking turns.	
Block Creation Throttling per node	No. A block creator will remain a block creator until another node starts an election.	Yes. Half of all nodes must create a block before the same node may create another block.	Yes. No node may create two blocks in a row.
Block Replication	Current and previous blocks (up to the snapshot) are replicated by the consensus protocol.	No. Only current block is replicated by the consensus protocol.	
Communication Overhead	Average communication overhead (Replication, voting and heartbeats).	Close to zero communication overhead (Data is attached to block headers).	Significant overhead (three phases of communication per block).
Node Connectivity	Leader is directly connected with all followers.	No extra connections required by consensus protocol.	Every node connects and shares messages with a subset of nodes using gossip.

Table 5.3 Overview of Consensus Protocol Functionality

5.3.4 Consensus Protocol Properties

Finality

During our discussion of DLTs, Section 2.1 briefly mentioned transaction **Finality**. This refers to the notion that once a transaction has been included in a block, its ordering is permanent and may not be reverted (due to another longer chain or otherwise). Finality is not *guaranteed* in Clique, since there is always the chance that a soft fork occurs at some (past) location in the chain. When this chain becomes the longest, nodes consider this fork as valid. Yet, there are robust restrictions in place from preventing this from happening, such as the restriction on nodes to only sign 1 out of `N` blocks. As long as the network is not compromised, we speak of *probabilistic* finality, meaning that as time approaches infinity, the chance of a block being reverted approaches 0. In practice though, it means that one should wait for a couple of confirmations (consecutive blocks). In Raft, an ordered block is immediately final. Although it is possible in Raft to have multiple orderings of proposed blocks, the network is unaware of those until they are committed. Likewise, IBFT 2.0 also has immediate finality due to its three-phase commit.

Crash Fault Tolerance

Another interesting property for permissioned blockchains is CFT, meaning the network will continue operating reliably while some nodes are unavailable. Luckily, all three consensus protocols are CFT, although they differ in their extent. Raft always requires at least a majority of nodes (votes/confirmations). Thus, strictly less than (but not exactly) 50% of the nodes are allowed to crash, i.e. minority. Clique also continuous operations while at most a minority of nodes are unavailable for the creation of blocks, since a node has to wait for `N/2` other nodes to sign before him. In IBFT 2.0, at least 2/3rd of all validators must reply before nodes continue to the prepare or commit phase. Consequently, at most one-third of the nodes may fail.

Byzantine Fault Tolerance

Slightly different from CFT is BFT, in which nodes may not only fail, but become Byzantine. As we have seen in Section 5.1.6, no matter how large the network, only a single Byzantine node is required to undermine a Raft network. Thus, Raft is not BFT. Clique and IBFT 2.0 are both BFT. Clique ensures a majority of nodes must have created a block before a node may create another block, to prevent Byzantine impact on the chain(s). One third of the nodes in IBFT 2.0 may be Byzantine, without undermining the three-phase commit.

Minimum Network Size

Each consensus protocol has implications on the minimum number of nodes that are required to run. Although perhaps less interesting to production networks, this information could be useful for testing and development purposes. For Raft, only a single node is required. Then, we only have a leader with no followers. Similarly, Clique also supports a cluster of only 1 validator, allowing it to continuously create blocks without having to wait on the turns of others. IBFT 2.0 requires more nodes, namely a minimum of 3. There is no obvious reason for this, besides that a single node may not validate two blocks in a row. Still, nodes wait for at least two other trusted validators before starting to create blocks.

Overview of Consensus Protocol Properties

Consensus Protocol Properties			
Concept or Feature	**Fabric/Raft**	**Besu/Clique**	**Besu/IBFT 2.0**
Finality	Immediate Finality.	Probabilistic Finality.	Immediate Finality.
Quorum	Majority (N/2+1).	1 for block creation, Majority (N/2+1) for voting.	Two-thirds (2N/3) for block creation, Majority (N/2+1) for voting.
Crash Fault Tolerance	Yes.		
CFT Failures Allowed (max)	Minority ((N-1)/2).	Minority ((N-1)/2).	One-third (N/3).
Byzantine Fault Tolerance	No.	Yes.	Yes.
Byzantine Nodes Allowed (max)	0.	Minority ((N-1)/2).	One third (N/3).
Minimum Cluster Size	1 Node.	1 Trusted signer.	3 Trusted validators.

Table 5.4 Overview of Consensus Protocol Properties

Chapter 6

Data Privacy

As mentioned earlier in Chapter 2, data privacy is often crucial for permissioned blockchains. Fabric and Besu both support data privacy mechanisms in similar but slightly different ways. This chapter discusses how both frameworks aim to keep information secluded from other participants to answer the sub-research question: "***How is data privacy achieved in both frameworks, how is this information stored, and what does this imply for data availability and integrity?***" Both frameworks' implementations are put in context with the knowledge we have gathered so far on public transactions, e.g. in terms of architectural changes or changes to the transaction flow. At the end of this chapter, we again provide a comparison.

Recall that Chapter 3 studied the architecture of both frameworks in great depth, and compared them on aspects such as network isolation and internetwork smart contract composability. Clearly, dividing collaborating participants over multiple disjoint (sub)networks results in data privacy, especially since we have seen both frameworks have very limited support in terms of transactions across these (sub)networks (at least, up until now). Since the implications of having isolated (sub)networks are evident from the architectural chapter, as they are analogous to an outsider's perspective on a permissioned network (that is, a non-participant cannot see the ledger state, transactions, nor participants), this concept will not be further discussed in this Chapter. Rather, we focus on the solutions provided by the frameworks to provide data privacy *within* a single (sub)network.

6.1 Keeping data private in Fabric

Fabric's mechanism for ensuring data privacy is called a Private Data Collection (PDC), which allows participants to keep only some key-value pairs private among a set of predefined participants. First, an in-depth explanation of this concept is provided by Section 6.1.1. To facilitate this process, some changes to the knowledge gathered so far are required. For one, the peer's architecture of the ledger is slightly adapted, as described in Section 6.1.2. Secondly, Fabric's public transaction flow is extended to include additional steps, as explained in Section 6.1.3.

6.1.1 Private Data Collections

A PDC can be defined as an isolated key-value storage that is only shared with a well-defined subset of channel participants. A PDC allows organizations to deploy a chaincode in which some operations may be performed using the publicly shared channel ledger, whereas other operations are applied to the ledger state of the PDC. As such, a single transaction can thus query and modify the public channel's ledger, *and* modify the PDC at the same time. This enables use cases in which some information should be validated by all members, whereas other information should be kept private. An example use case is described below.

Let us consider a use case in the mining supply chain industry for tracking resources. After resources have been mined, it may switch ownership multiple times before the resource is processed in a consumer product. Responsible businesses may find it important to only purchase resources that provably originate from mines with a good reputation of not having child labour. Yet, when the supply chain is long and complex, it becomes difficult to track whether the resources originate from a reputable source. For the sake of this example, we refrain from the discussion on linking physical resources with digital ledger information. In this example, the resource identifier, origin and ownership listed on the public ledger, but the purchasing price and quantity in PDCs. By hiding the purchasing price and quantity, the seller prevents other organizations from leveraging this information for more bargaining power (such as demanding a lower price), while the network can still verify that a product's origin has not been changed (i.e. maliciously adapted) over time. Figure 6.1 illustrates this example. PDCs of Supplier 2, 3 and the end consumer are omitted for clarity.

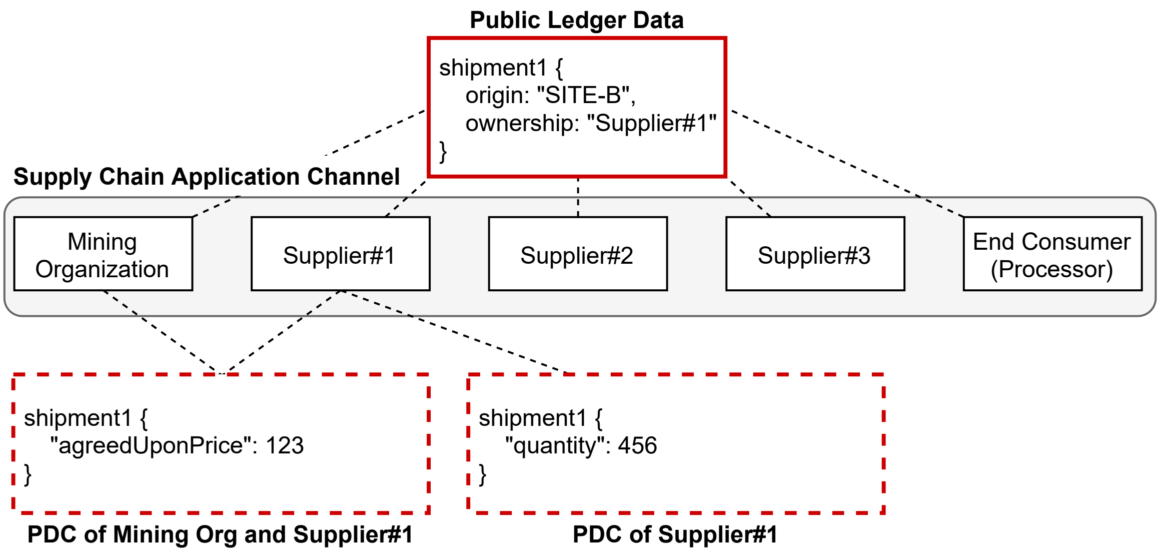


Fig. 6.1 Example data privacy setting

Keeping pieces of information private in a key-value storage of a subset of participants raises an important question: How meaningful is the private information, if it cannot be seen nor verified by other participants? In other words, there should be benefits in terms of verifiability to recording private data in a PDC rather than an off-chain agreement. To that end, a transaction involving private data always adds a hash of the private data in a public transaction. This public transaction is added to a block and distributed to all participants in the channel. If any member of a PDC decides to make (some of the) private information public, all channel participants are able to verify whether the publicly available hash matches the private data. Alternatively to making it public to all channel participants, private data could be shared with a single organization that one wants to transact with. This allows a transacting organization to verify the private information without being a PDC-member, e.g. a purchaser can provably verify the quantity of resources before placing a bid.

Parameters and Features of Private Data Collections

In practice, these collections are defined in a JSON-file as a part of the chaincode. Thus, modifications to the PDC must be approved by the same organizations as is defined by the administrative policy of the chaincode itself. An example PDC configuration is provided in Listing 6.1 and introduces other aspects and features discussed next.

```
[
 {
   "name": "MiningOrgAndSupplier1PDC",
   "policy": "OR('MiningOrg.member', 'Supplier1.member')",
   "requiredPeerCount": 1,
    "maxPeerCount": 1,
    "blockToLive":5,
    "memberOnlyRead": true,
    "memberOnlyWrite": true,
    "endorsementPolicy": {
      "signaturePolicy": "AND('MiningOrg.member', 'Supplier1.member')"
    }
  },
   {
    "name": "Supplier1PDC",
    "policy": "OR('MiningOrg.member')",
    "requiredPeerCount": 0,
    "maxPeerCount": 1,
    "blockToLive":0,
    "memberOnlyRead": true,
    "memberOnlyWrite": true
},
  // ...
]
```

Listing 6.1 Example Private Data Collection definition

A first observation should be that it is not mandatory for private data to be distributed to all PDC-members, as denoted by `requiredPeerCount` and `maxPeerCount`. These parameters define how many other peers an endorser passes the resulting readset and writeset to, before the client receives a response. `maxPeerCount` denotes how many other peers it tries to disseminate the data to, whereas `requiredPeerCount` denotes how many of these disseminations should succeed before the endorsement is considered successful. This naturally implies that some organizations may not receive the private data and, therefore, ledger state of different PDC members may be inconsistent. If a PDC member is asked to endorse a transaction while missing the private data, it tries to solve this issue by pulling it from other PDC members.

Another aspect is that nothing enforces the indefinite persistence of private data. Since private data is not recorded on the blockchain itself, a peer may lose this information, for instance, when its hard drive crashes and it starts resyncing from the genesis block. While we already saw a peer tries to pull missing private data, there is no guarantee it will succeed. When all other peers do not have this information (anymore) as well, or there are no other peers in the PDC (i.e. a 1-organization PDC), the information is lost forever. The PDC-definition mentioned in Listing 6.1 actually contains a property that causes this phenomena to occur deliberately, namely the `blockToLive`-property. This value denotes the number of blocks, after which the members of the PDC deliberately delete the private data. For our example, we see that the agreed-upon price is automatically removed from `MiningOrgAndSupplier1PDC` after 5 blocks. Of course, deletion of private data is not required and can be disabled by setting the `blockToLive` to 0, as we see with the *quantity* information in `Supplier1PDC`. Although this approach could be useful for enforcing on-chain agreement (through the endorsement policy) for data that can be discarded afterwards, one should wonder if it makes sense to record such volatile and short-lived information on the ledger in the first place.

Lastly, we have the `memberOnlyRead` and `memberOnlyWrite` parameters. As the name suggests, these indicate whether only a member of the PDC is able to read from and write to the private state. This might be counter-intuitive, since setting these to false would allow a non-member to read or write *private* data it is not supposed to have. Instead, this is meant to be used in combination with chaincode-level ABAC. For instance, a potential buyer could be granted permission to query only one specific asset's details directly (rather than comparing its hash). Or, when no information is public and the entire assets are passed using PDCs, a seller might write the asset to another organization's PDC directly. In any case, the organizations of the PDCs can still be in control over what external organizations may read and write by having programmatic runtime checks in place in the chaincode.

Best Practice

Our previous example (Figure 6.1) showed a setting in which there was a PDC shared among a group of organizations, and a PDC shared with only a single organization. Both techniques show a fundamentally different way of thinking about private data: do we create many PDCs for every possible combination of transacting organizations, or do we create a PDC per organization? The documentation of Fabric encourages users to use the latter approach: "*consider using a smaller number of private data collections (e.g. one collection per organization, or one collection per pair of organizations)*"[1]. To that end, Fabric v2.0 supports *implicit PDCs*, which means that every organization in a channel automatically has its own PDC. An advantage of this approach would be that when an organization joins or leaves a channel, the chaincodes do not need to be updated to add or remove the organization from PDCs. Since the endorsement policy of PDCs with only one organization only requires that single organization to endorse a transaction, validating hashes becomes an even more important step to ensure that the private data was not modified.

6.1.2 Change(s) to the peer's internal architecture

While discussing Fabric's architecture, we saw that each peer's ledger maintains a blockchain and a key-value storage for each channel. In order to support PDCs, two additional stores are required. First of all, a separate store for each individual PDC is required to store the actual private data without having conflicting keys. Secondly, a *transient data store* is required. As we will see next during the transaction flow discussion, private data that is received during an endorsement process should be temporarily remembered until a transaction (and its private hash) is actually committed. This information may not be committed to the state database directly, because the transaction has not been validated yet (it may not have enough endorsements, it may have a conflicting readset, et cetera). Therefore, this private data is temporarily stored in this transient data store, until it encounters the corresponding transaction in the block, which is when it will (attempt to) apply the changes if the transaction is valid. Figure 6.2 illustrates how the new ledger's state is organized with PDCs.

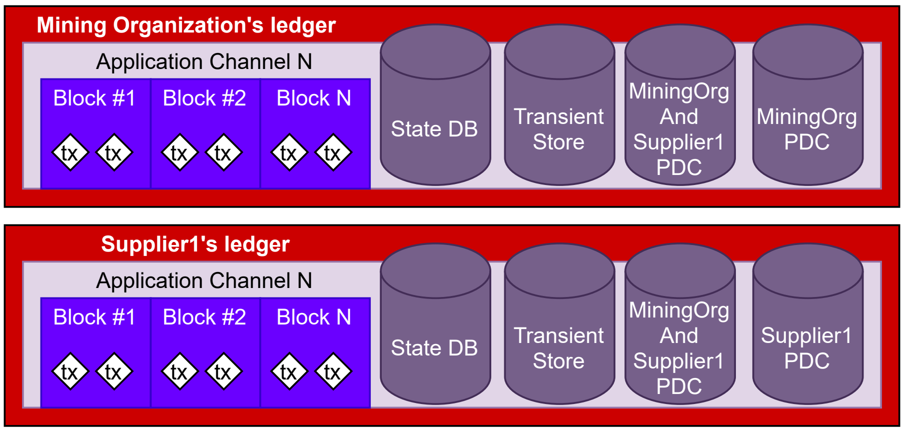


Fig. 6.2 Peer's ledger with Private Data Collections

6.1.3 Change(s) to the public transaction flow

Recall from Section 4.1 that blocks go through the ordering service and are made available to all peers of a channel. Thus, private data has to be disseminated before a block is created and distributed through the ordering service. Moreover, even though the ordering service is expected to ignore a block and transaction's contents, an OSN with access to private data is still a security risk. Therefore, private data in Fabric is disseminated peer-to-peer. To explain how this private transaction flow differs from the public transaction flow, the reader is once more guided through the high-level process from creating a

[1] https://hyperledger-fabric.readthedocs.io/en/release-2.2/private-data/private-data.html#sharing-private-data

transaction to committing it. Figure 6.3 shows the high-level overview of the private transaction flow. This figure is an adaptation of the public transaction flow given in Figure 4.1.

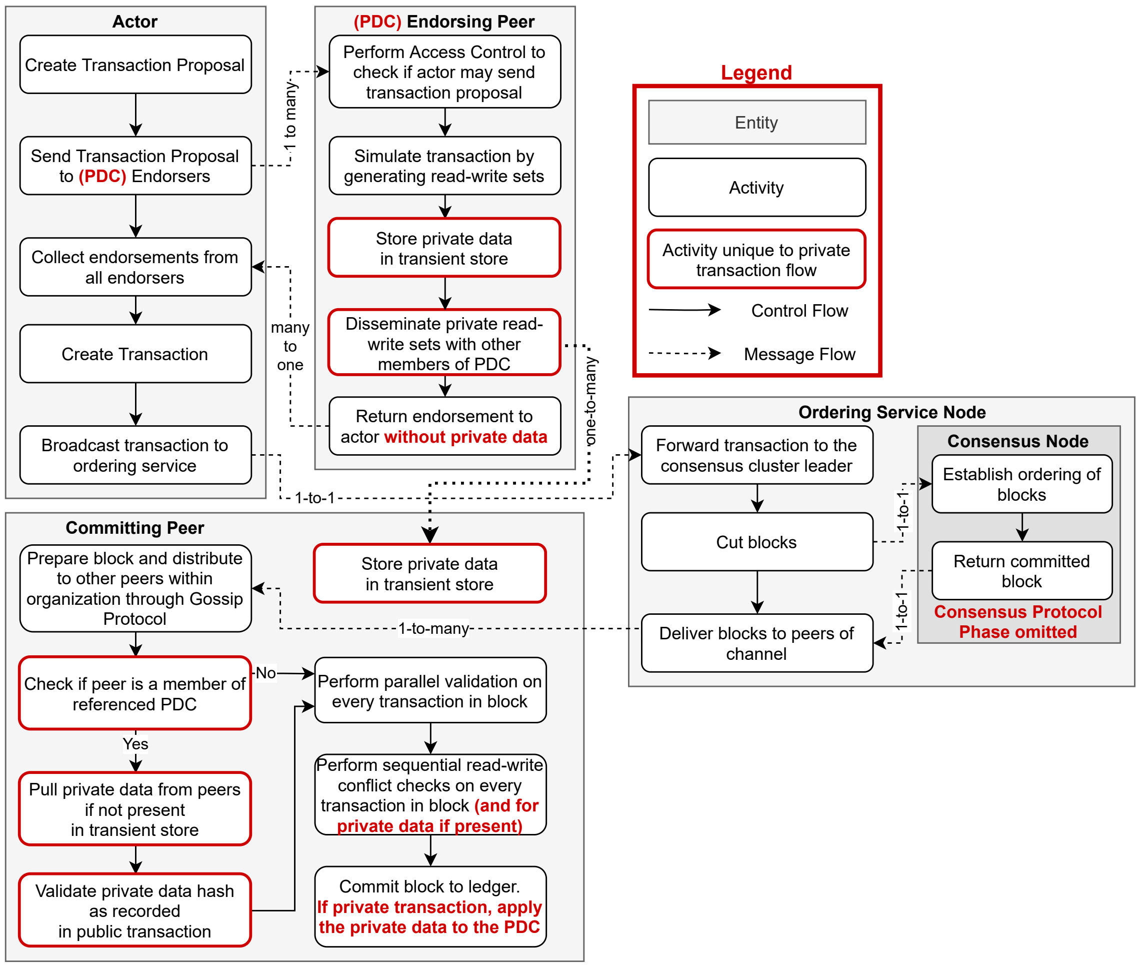

Actor
Create Transaction Proposal
Send Transaction Proposal to (PDC) Endorsers
Collect endorsements from all endorsers
Create Transaction
Broadcast transaction to ordering service
1 to many
many to one
(PDC) Endorsing Peer
Perform Access Control to check if actor may send transaction proposal
Simulate transaction by generating read-write sets
Store private data in transient store
Disseminate private read-write sets with other members of PDC
Return endorsement to actor without private data
one-to-many
Legend
Entity
Activity
Activity unique to private transaction flow
Control Flow
Message Flow
Ordering Service Node
Forward transaction to the consensus cluster leader
Cut blocks
Deliver blocks to peers of channel
Consensus Node
Establish ordering of blocks
Return committed block
Consensus Protocol Phase omitted
1-to-1
1-to-1
1-to-1
1-to-many
Committing Peer
Prepare block and distribute to other peers within organization through Gossip Protocol
Store private data in transient store
Check if peer is a member of referenced PDC
No
Yes
Perform parallel validation on every transaction in block
Pull private data from peers if not present in transient store
Perform sequential read-write conflict checks on every transaction in block (and for private data if present)
Validate private data hash as recorded in public transaction
Commit block to ledger. If private transaction, apply the private data to the PDC

Fig. 6.3 Fabric's Private Transaction Flow

Like in the public transaction flow, the process starts once an actor intends to create a transaction. It will create this transaction in the same way as a public one, namely by creating a proposal and sending it to endorsers. It's important that the actor submits it to members of the PDC, rather than just any endorser. After the endorser has performed ABAC and simulated the transaction, it locally stores the private data in the transient data store, until the private data is retrieved later. The private data is then disseminated to a (subset of) peers that are a member of the same PDC, who also store this in their transient data store. If the private data was successfully disseminated to at least `requiredPeerCount` peers, it returns an endorsement to the client. This endorsement, however, does *not* contain the actual private data, because the endorsement is to be ordered into blocks later. Instead, it contains a hash of the private data. Therefore, if the client wants to get private data from a node, it has to wait until the transaction is ordered and committed.

Now, the client waits until it has received a sufficient amount of endorsements. As was explained in Section 3.1.9, the endorsements required are defined by the endorsement policy. If the PDC has its own endorsement policy defined (as shown in the example of Listing 6.1), that endorsement policy takes precedence over the chaincode's or channel's policy. The endorsements are gathered in a transaction, which is broadcast to the ordering service. Note that this operation is safe at this point, since it does not contain any private data. After the block is ordered, it is (indirectly) distributed to all committing peers. If the peer is not a member of the PDC, it will commit the block as usual. Otherwise, it checks if it already has the private data in its local transient data store, otherwise it tries to pull this information from other peers of the same PDC. For safety measures, it computes a new hash over the private data of its transient store and compares this with the hash recorded in the public transaction. Only if it matches, it considers the transaction and private data as valid and continues the transaction processing as usual. Only after the public transaction is appended to the ledger, the private data is written to its local PDC store.

6.2 Keeping data private in Besu

Besu supports data privacy by allowing nodes to execute private transactions that invoke private smart contracts among a group of participants. These transactions and smart contracts live in an isolated key-value state from the ledger's main world state. We previously showed that the main ledger contains a `PrivateWorldStateStorage`, as we have seen in Listing 3.1 and Figure 3.13. To recap, the private world state contains a mapping of `address->account` for tracking an SCA's balances, `address->code` for storing a SCA's smart contract, and `address->storage` for storing the smart contract's state. Every node contains exactly one private state storage. In a single channel, there could be multiple groups of participants that transact privately. Each of these groups requires its own private world state. Yet, we have seen that the ledger only contains a *single* private world state. In practice, Besu logically divides this private state storage up in namespaces such that each group's state is isolated from the state of other groups. This is analogous to adding a prefix for each group, e.g. `group1-...`, `group2-...`.

Before a transaction may be executed and its results may be written to this private world state, the private data needs to be shared with and stored by all other participants of this private group. Distribution and storage of this private transaction occurs *externally* to the Besu node itself. Namely, it is sent to an *Orion* node which has to be introduced to the network architecture, as explained in Section 6.2.1. After that, a public transaction is distributed and included in blocks like a normal transaction. This private transaction flow is explained in more detail in Section 6.2.2. After having discussed the changes to the architecture and transaction flow, we take a closer look at the private transaction itself and its functionality from a practical point of view. In fact, transactions in Besu can be used to manage private groups in different manners, as explained in Section 6.2.3. Lastly, as Besu has been presented as a shell around the original Ethereum Protocol up until now, we discuss the MainNet compatibility of private transactions in Section 6.2.4.

6.2.1 Adding an Orion-node to the architecture

Orion is a software component by ConsenSys[2] for storing and distributing private transactions. Every node in the network that wishes to participate in private transactions must have an Orion node deployed side-by-side with its Besu node, as shown in Figure 6.4. From this image, we see that communication from Besu nodes to Orion nodes is one-way. Namely, Orion nodes act like a service for Besu nodes: an Orion node is always invoked one-way by Besu nodes, and never vice versa. Like we saw earlier with Besu nodes, an Orion node discovers other Orion nodes through a list of bootnodes.

Most importantly, an Orion node has to expose `send` and `receive` functionality to Besu nodes. Listing 6.2 exemplifies a payload for both types of requests. The `send`-request is used to submit a serialized `PrivateTransaction` object to the Orion node, i.e. a transaction that only a group of participants execute on a smart contract. In addition, the request defines the target group of all other Orion nodes that the request should be passed to as well. The Orion node stores the `PrivateTransaction` locally and disseminates it to all other Orion nodes in the target group. Only when *every single* Orion node referenced has acknowledged storing the data, the node to which the client request was made returns a hash of the data. Since the same `PrivateTransaction` could theoretically be sent to two different and possibly overlapping private groups, the hash is also computed using the target group, i.e. `hash(privTx, targetGroup)`. This hash is called the privacy marker. When a Besu node later on needs to fetch this `PrivateTransaction` again, it can call the `receive`-function on the Orion node. To do so, the Besu node has to supply the privacy marker (`key`) that was returned by the Orion node earlier when the data was submitted.

[2] https://github.com/ConsenSys/orion

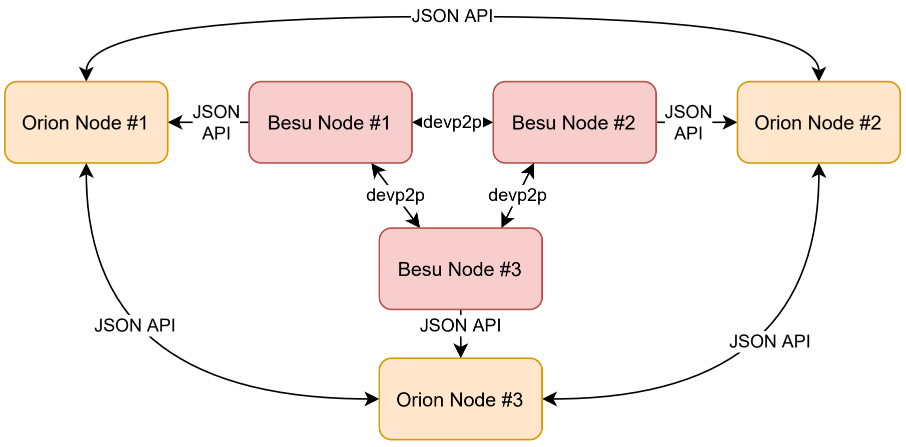


Fig. 6.4 Network of Besu nodes in combination with Orion Nodes

```
// POST-Request payload for sending a Private Transaction
{
    "payload": "SGVsbG8sIFdvcmxkIQ==",
    "to": ["YE5cJRJYTR04XFo7yuAi/0K9DwjySGjsHB2YrFPnJXo="]
}

// POST-Request payload for receiving a Private Transaction
{
    "key": "wS+RMprLKIuCaHzOBfPeHmkJWUdOJ7Ji/9U3qj2jbXQ="
}
```

Listing 6.2 Sending and Receiving Private Transactions from Orion

Worth mentioning is that during the start up of a Besu node, it is supplied with an argument with the address of its affiliated Orion node (if privacy is enabled). Yet, Orion nodes do not keep track of their corresponding Besu node. Moreover, an Orion node does not have any form of access control or permissioning. As a result, it means that a private transaction can be retrieved by anyone with knowledge of the privacy marker, while being able to establish a connection with the node. The updated architectural diagram including Orion nodes is shown in Figure 6.5.

6.2.2 Private Transaction Flow

To use Besu's data privacy features, we again start by the actor. Instead of creating a regular transaction using `eth_send(Raw)Transaction` as we have seen in Section 3.2.6, a transaction intended for data privacy is a different kind of transaction. Namely, *every* single private transaction must include a field that references the private group. This information is required for all types of transactions, i.e. (private) contract creation transactions and (private) message calls. Since Besu decided to adapt the format of the private transaction itself, the actual smart contracts remain completely unchanged. In other words, a smart contract of a private group could have been deployed without any changes as a public contract instead. For private transactions, just like public transactions, the same EthSigner mechanism is in place, as discussed in Section 3.2.6. Namely, a Besu node only accepts a signed private transaction as input. To allow actors to send a transaction without signing it with their private key, they can call the `eth_sendTransaction`-endpoint of an EthSigner instance, which makes a `eth_sendRawTransaction`-call to the actual Besu node.

Once a Besu node receives a private transaction on the aforementioned endpoint, it will handle this transaction differently than a public one. As a first step, it will call the `send`-method on the Orion node that is configured for that Besu node. Only if the Orion node has successfully stored the transaction locally and distributed it to *all* other Orion nodes referenced in the private transaction, the operation succeeds and a key is returned to the Besu node. This key is actually a hash of the

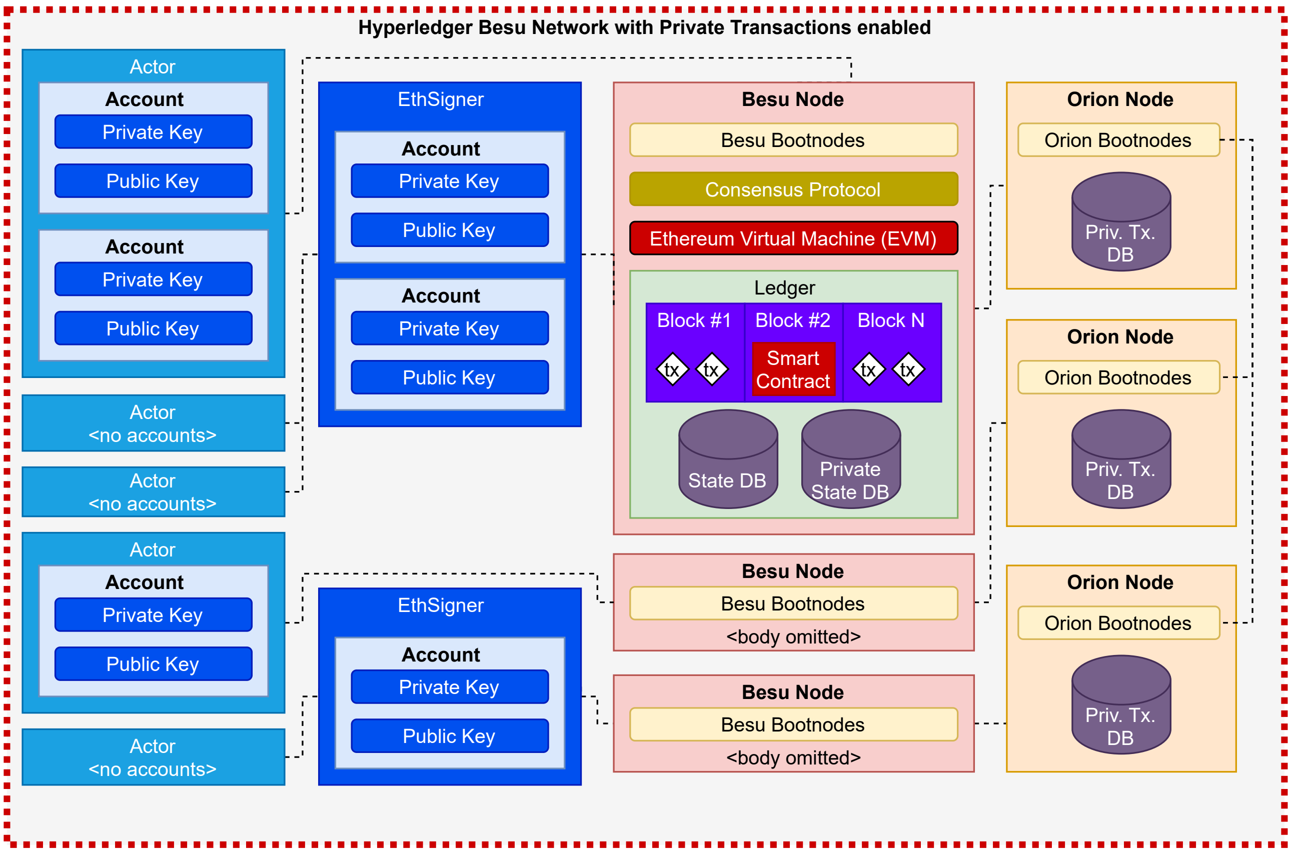


Fig. 6.5 Updated Network Architecture for Data Privacy

private transaction and maps to the unhashed private transaction itself, as explained in Section 6.2.1. At this point in time, the private transaction is not inspected nor executed (it is only stored on Orion) and can be forgotten by the Besu node, at least until the Besu node receives a block with a marker.

After the private transaction was successfully distributed, the returned key (or hash) is used to create a *Private Marker Transaction*. In contrary to a private transaction which has a different structure due to additional fields, the private marker transaction has the same format as defined in the Ethereum protocol. What makes this transaction different from others, is its `to` field. Instead of referencing an EOA or SCA, the `to`-address is set to a reserved and fixed address. By always using the same address, like a flag, all Besu nodes recognize the transaction as a private marker and thus, know to handle this transaction differently. Next, the creator of the private marker transaction needs to ensure that all participants in the private group know what actual private transaction, as stored on Orion, is associated with the marker. Therefore, the hash of the private transaction is recorded in the private marker transaction's payload. This information can be distributed and recorded safely by all nodes in the network, even those not included in the private group, as a node cannot reconstruct the actual private transaction from its hash.

The private marker transaction is added to the node's local transaction pool and distributed peer-to-peer like any other transaction. Once it receives a block with the to-address set to the flag, a `retrieve`-request is made to the node's Orion instance using the hash recorded in the payload. If the Orion node is not part of the private group, it returns that it does not have a private transaction with that key, in which case the Besu node does not process it any further. Otherwise, if the Orion node has a private transaction stored under that key, it is returned to the Besu node. The Besu node now executes the private transaction using a unique state (key-value space) for that private group. During the execution of a private transaction, it may call a public smart contract or another smart contract of the same private group. For obvious reasons, it cannot call a private transaction of another private group('s namespace). Although private smart contracts may read any value from any

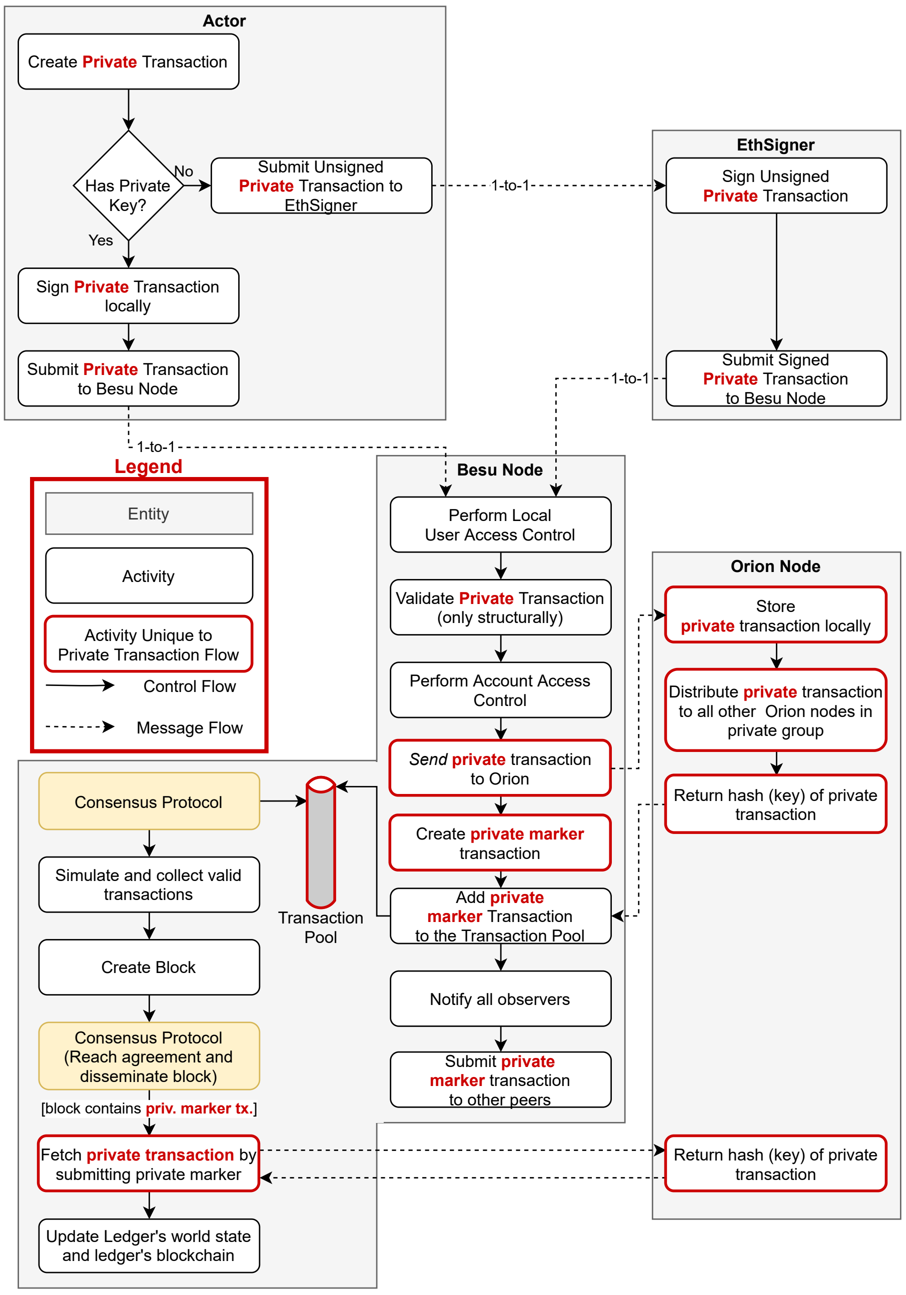

Actor
Create Private Transaction
Has Private Key?
No
Yes
Submit Unsigned Private Transaction to EthSigner
Sign Private Transaction locally
Submit Private Transaction to Besu Node
1-to-1
EthSigner
Sign Unsigned Private Transaction
Submit Signed Private Transaction to Besu Node
1-to-1
1-to-1
Legend
Entity
Activity
Activity Unique to Private Transaction Flow
Control Flow
Message Flow
Besu Node
Perform Local User Access Control
Validate Private Transaction (only structurally)
Perform Account Access Control
Send private transaction to Orion
Create private marker transaction
Add private marker Transaction to the Transaction Pool
Notify all observers
Submit private marker transaction to other peers
Orion Node
Store private transaction locally
Distribute private transaction to all other Orion nodes in private group
Return hash (key) of private transaction
Return hash (key) of private transaction
Consensus Protocol
Transaction Pool
Simulate and collect valid transactions
Create Block
Consensus Protocol (Reach agreement and disseminate block)
[block contains priv. marker tx.]
Fetch private transaction by submitting private marker
Update Ledger's world state and ledger's blockchain

Fig. 6.6 Besu's private transaction flow

public smart contract by calling it explicitly, the private transaction processor is unable to modify the public ledger's state. The resulting Transaction Flow is visualized in Figure 6.6.

6.2.3 A closer look at Private Transactions and private groups

So far, we have explained that a private transaction has a different structure than a public transaction by referencing a private group. In fact, there exist multiple manners in which private groups are referenced and managed, explained next. First, EEA-compliant private transactions are discussed, after which we discuss some extensions of Besu that are not defined by the EEA standard. Lastly, flexible private groups are briefly mentioned.

EEA-Compliant Private Transactions

A private transaction introduces an additional field: `privateFor`, as can be seen in Listing 6.3[3]. This field should be set to an array of Orion node addresses to distribute the transaction to. Note that although a transaction is *submitted by* an *EOA* and executed on *Besu nodes*, the actual private transaction references the addresses of *Orion nodes*. This means that we now have three different types of addresses/identifiers. Although perhaps confusing at first, the reason for this is that Orion nodes run their own peer-to-peer network completely independent from actors and Besu nodes. This is simply a design choice that allows Orion nodes to distribute the data themselves, instead of requiring Besu nodes to distribute this data. Yet, it does mean that all members of the private group need to know all Orion addresses and pass this to each private transaction every time.

```
{
    "jsonrpc":"2.0",
    "method":"eea_sendTransaction",
    "params":[
        {"from": "0xfe3b557e8fb62b89f4916...",
        "data": "0x608060405234801...",
        "privateFor": ["g59BmTeJIn7HI...", "Dbvbt2eyP0Ii60aDDw..."],
        "restriction": "restricted"}
    ],
    "id":1
}
```

Listing 6.3 Example of sending an EEA private transaction

Another aspect of this relates to private group management. At all times, the `privateFor` denotes all members of the private group. Implementation-wise, the Orion node passes this list of members through a function to get a group identifier. This group identifier is used to separate values from one private group with values of another. Consequently, adding or removing a member from the `privateFor` field would result in a different group identifier, and thus, a different namespace and private state. In other words, a small change to the `privateFor` field would be seen as a completely different (and new) group. Consequently, once a group has been defined, it cannot be changed at all. This would likely be a major disadvantage for permissioned networks for which the network's composition and members are expected to be dynamic.

Besu-Extended Private Transactions

In order to make private groups more user friendly, Besu introduced the concept of group aliases. Instead of passing in a (potentially long) list of addresses of Orion nodes, only an alias needs to be provided, as seen in Listing 6.4. A privacy group can be created by calling the `createPrivacyGroup` method on an Orion node, by specifying the group id and list of Orion addresses as members. This change not only makes it more convenient to create a private transaction by only specifying a simple identifier, this group identifier is now actually decoupled from the member list. Instead of a list of members mapping to an identifier, an identifier now maps to a list of members (i.e., the mapping is inversed). This would technically make it possible to add or change members without the risk of colliding namespaces. Yet, changing the group structure is unfortunately (still) unsupported.

```
{
    "jsonrpc":"2.0",
    "method":"eea_sendTransaction",
    "params":[
        {"from": "0xfe3b557e8fb62b89f4916...",
        "data": "0x608060405234801...",
        "privacyGroupId": ["myPrivateGroup"],
        "restriction": "restricted"}
    ],
```

[3]For readability purposes, the request to an EthSigner instance is provided. As explained before, Besu only accepts signed requests.

```
    "id":1
}
```

Listing 6.4 Example of sending a Besu-Extended private transaction

Flexible Private Groups

Although private groups are rigid and set in stone when they are created, the developers are working on a solution to have flexible group membership, by defining the members in an on-chain smart contract. Most importantly, it is expected to support functionality for adding and removing members of the private group. At this point in time, though, this feature is marked as an "early access feature" that should not be used in production networks[4], according to the documentation. In addition, this feature was marked as "ongoing and support for [...] flexible privacy groups isn't complete"[5], was reported to contain race conditions[6], and has not been updated since (while checking the corresponding file history and changelog of the GitHub repository).

6.2.4 MainNet Compatibility

Now that we have seen how data privacy is achieved in Besu, one might wonder if data privacy can be activated on the Ethereum MainNet. Especially, since only the private marker transaction is shared and recorded on the main ledger which follows the normal transaction structure. It is evident that the architects of Besu tried to implement privacy while not modifying any of the core mechanics of the Ethereum Protocol. In an experiment of mine, I was able to submit a private marker transaction on the Ethereum Ropsten test network, which follows the same consensus protocol as the MainNet. Althoguh this was successful, showing it can be done, there is still an issue with using the current implementation data privacy on the Ethereum MainNet, discussed next.

Private transactions in Besu actually require all transactions to be immediately final, as written in the documentation: "*Do not use private transactions in production environments using consensus mechanisms where forks occur*"[7]. As our analysis in Chapter 5 showed, only IBFT 2.0 provides deterministic block finality, meaning Clique and PoW are excluded from being used in the current version of Besu. Therefore, even though private transactions are mostly distributed, executed and stored independently of public transactions, they cannot be used in conjunction with the Ethereum MainNet network due to its PoW consensus protocol with stochastic finality. After looking at the source code of Besu, the reason for this limitation becomes clear. When a new block is received that is a fork, only the public world state is rolled back when imported. There does not exist any code that rolls back the private world state to a parent block, as is done on the public state. Since the private world state uses an implementation (`WorldStateKeyValueStorage`) that implements the exact same interface (`WorldStateArchive`) as the public world state implementation (`BonsaiWorldStateKeyValueStorage`), the rollback functionality is technically already supported, but simply not activated for the private state.

To conclude, private transactions on the Ethereum MainNet are currently not supported in Besu due to being incompatible with nondeterministic consensus protocols: Rollback functionality is never called on private transactions. This limitation could possibly be overcome in the future without too much effort. Namely when Besu adapts the `BlockValidator` by supporting rollbacks or when Ethereum would move to a deterministic consensus protocol, the latter being unlikely in the short term. Although this change will resolve the current and most prominent issue with private transactions, they may admittedly not solve *all* issues as other problems might pop up. Thus, this could be future work to look into.

[4]https://besu.hyperledger.org/en/stable/HowTo/Use-Privacy/Use-FlexiblePrivacy/
[5]https://github.com/hyperledger/besu/pull/1032
[6]https://github.com/hyperledger/besu/issues/2046
[7]https://besu.hyperledger.org/en/stable/Concepts/Privacy/Privacy-Overview/

6.3 Findings

In this chapter, we have seen how both frameworks implement data privacy within a single (sub)network. Interestingly, both frameworks try to solve the problem of data privacy in the same way on a conceptual level: private data is distributed peer-to-peer among the members of a private group, while creating a hash of the private data on the public ledger. When that hash is encountered during block processing, the private data is applied *only* by members of the private group or collection. It is remarkable how similar Besu and Fabric's approaches are, yet their implementations differ substantially. In this comparison, we start by going over some of the more technical differences in how their implementations vary. Then, their most important implications are discussed for data integrity and data availability, since private data is never stored on the actual ledger itself for both frameworks.

6.3.1 Technical Differences

Distinguishing private from public transactions

In order to achieve data privacy, nodes require knowledge of whether they should execute a given transaction on the public state, or on the private state. Both frameworks distinguish between the two different types of operations in very different ways. For Fabric, the actual smart contracts use different operations for private data (i.e. `getPrivateData` and `putPrivateData`) than the operations for public data (i.e. `getStringState` and `putStringState`). By modifying the interface of chaincode operations, the actual private transaction proposals do not differ from public transaction proposals. For Besu, this is the other way around. Private and public smart contracts *could* be completely equivalent, meaning the same smart contract could be deployed publicly or as part of a private group. Besu does not have different public and private operations because it defines the distinction using transactions: a transaction for data privacy mentions the private group, which is how the node knows it should execute the operation in a different manner.

Private data and its distribution

Besides the execution of private transactions, we can look at the type of private data which the frameworks actually distribute over the network. For Besu, distributed data are actually the private transactions themselves. These private transactions are forwarded to a Besu node's corresponding Orion node, which stores and distributes these private transactions with others. In Fabric, like with public transactions, the actual private state is disseminated in terms of a readset and writeset. This private state is is distributed directly between Fabric nodes through gossip. Both frameworks encourage this private data, albeit a private transaction or private state, to be sent using TLS.

Simultaneous public and private state modification

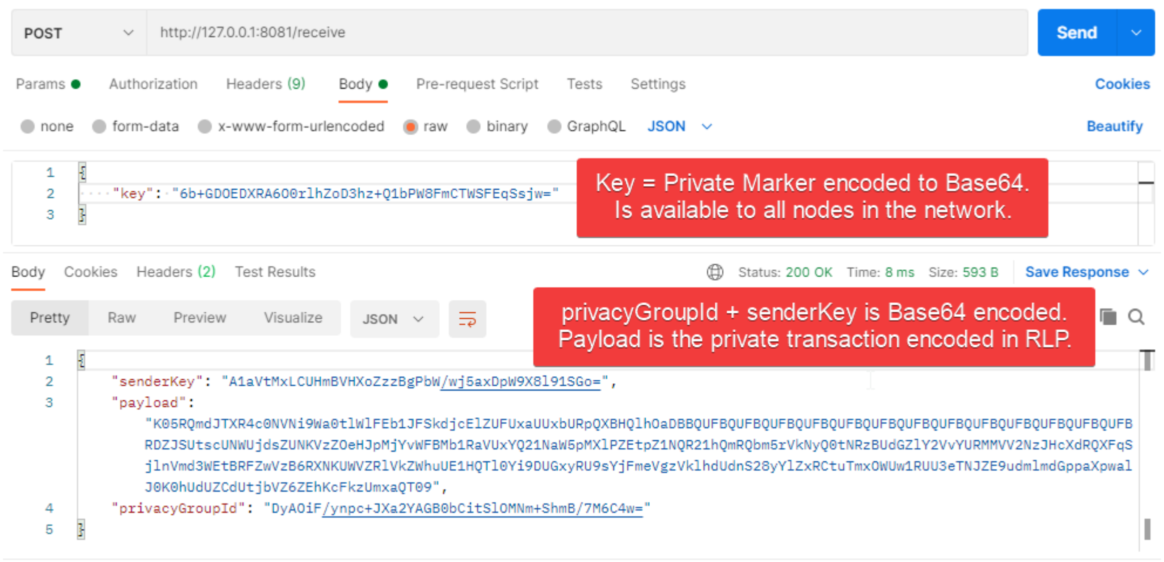
The two aspects discussed above, i.e. how private data is defined and how a node distinguishes between public and private operations, are not *just* technical differences meaning that developers create and use contracts and transactions in a different manner, they have some functional consequences as well. First, consider updating both the ledger's public state and private state simultaneously. For Fabric, this functionality is supported because of the combination of (1) not distinguishing between public and private transactions, and (2) by disseminating private state rather than the transaction itself. By using privacy-specific operations in a smart contract, a node can easily split up public and private state. For Besu, updating public and private state in the same transaction is not possible, since private transactions are executed after ordering on private smart contracts. In addition, a (private) transaction may only update the state of the privacy group it is deployed in, so even though a private smart contract may call a public smart contract such as for read requests, only the private smart contract's state is updated.

Interaction between private groups

Another functional consequence of how both frameworks define private data and interact with them, relates to smart contract interaction between private groups. In other words, a (sub)network might have multiple private groups that usually share everything among themselves, but occasionally have to transact with other private groups as well, e.g. when an asset needs to be transferred from one group to the other. Because a Fabric chaincode can query the public ledger in the same transaction as private state in the same call, such functionality can be implemented as seen in the example of Section 6.1.1. Since Besu can only update either the public or private state, it is evident such intergroup operations cannot be performed. For Besu, all private groups should be considered as their own isolated namespace, with the public state as a read-only supplemental library.

Access to private data by non-members of a private group

Although private data is originally designed to only be shared between members of a private group, both frameworks allow non-members of private groups to retrieve this information directly from private storage in some way. For Fabric, the `memberOnlyWrite` field could be set to false, allowing anyone to query the private state of another node. When used in combination with access control in the chaincode, access by non-members can be programmed. Thus, this is only an optional and more advanced feature that can be enabled if desired. For Besu, private *state* is never distributed (it is only computed locally), but the private *transaction* is distributed, as we have seen already. Yet, Orion has no access control at all! Recall that when sending a private transaction to Orion, a hash is returned as privacy marker. This marker is included in the payload of a public transaction, which all nodes receive. When a node sees a marker transaction with the privacy reserved address, it attempts to fetch the private transaction from the Orion node. To fetch this information, only the marker hash is required. Thus, anyone with this information and the address of the Orion node can fetch the private data, as demonstrated in Figure 6.7. Although it does not contain private state, one can easily reconstruct the called function and arguments, leaking crucial information. Moreover, the Orion node cannot be placed behind a firewall, nor can its address be hidden from other nodes, as it (by design) discovers and connects to all other Orion nodes (through bootnodes), as is required for private data dissemination. *Only* when all private groups are completely disjoint with not a single member in two groups, it would be safe to put Orion nodes behind a firewall. Then, only other Orion nodes of the same private group could be whitelisted. Since the smart contracts themselves are recorded in a similar way, one would be able to reconstruct the entire private data collection by fetching the contract and all transactions.



```
http://127.0.0.1:8081/receive
```

```
{
    "key": "6b+GDOEDXRA6O0rlhZoD3hz+Q1bPW8FmCTWSFEqSsjw="
}
```

```
{
    "senderKey": "A1aVtMxLCUHmBVHXoZzzBgPbW/wj5axDpW9X8l91SGo=",
    "payload":
        "K05RQmdJTXR4c0NVNi9Wa0tlWlFEb1JFSkdjcElZUFUxaUUxbURpQXBHQlhOaDBBQUFBQUFBQUFBQUFBQUFBQUFBQUFBQUFBQUFBQUFBQUFBQUFB
        RDZJSUtscUNWUjdsZUNKVzZOeHJpMjYvWFBMb1RaVUxYQ21NaW5pMXlPZEtpZ1NQR21hQmRQbm5rVkNyQ0tNRzBUdGZlY2VvYURMMVV2NzJHcXdRQXFqS
        jlnVmd3WEtBRFZwVzB6RXNKUWVZRlVkZWhuUE1HQTl0Yi9DUGxyRU9sYjFmeVgzVklhdUdnS28yYlZxRCtuTmxOWUw1RUU3eTNJZE9udmlmdGppaXpwal
        J0K0hUdUZCdUtjbVZ6ZEhKcFkzUmxaQT09",
    "privacyGroupId": "DyAOiF/ynpc+JXa2YAGB0bCitSlOMNm+ShmB/7M6C4w="
}
```

Fig. 6.7 Demo of Orion node exposing private data without access control.

Overview of Technical Differences on Data Privacy

Technical Differences on Data Privacy		
Concept or Feature	**Fabric**	**Besu**
Changes required to Private Smart Contract compared to Public SC	A private smart contract differs from a public smart contract by explicitly referencing and modifying private data collections.	A private smart contract can be equivalent to a public smart contract, without any changes.
Changes required to Private Transaction compared to Public Tx	A transaction proposal involving data privacy can be equivalent to a transaction proposal without data privacy, without any changes.	A private transaction differs from public transactions by explicitly mentioning the private group.
Private Data Distributed	Private state (readset and writeset).	Private transaction (without state).
Private Data Distribution Technique	Peer-to-peer gossip, between Fabric nodes.	Direct Peer-to-peer, using external components (Orion nodes).
Simultaneous Modification of Public and Private State	Supported.	Unsupported. A transaction may only modify public state, or only the private state, but not simultaneously.
Intergroup Smart Contract Execution	Supported, by using the hashcode in the public channel to verify data validity, or alternatively by requesting the data if access control is used.	Unsupported
Private Data can be requested by:	Only members of the PDC by default, otherwise programmable access control in the chaincode.	Anyone with Orion address and private marker, there is no access control (!).

Table 6.1 Overview of Technical Differences on Data Privacy

6.3.2 Implications for Data Integrity

We already saw that the developers of permissioned blockchains are typically faced with a trade-off between security (or integrity) of the data and their level of privacy, as explained in Section 2.2.2. As this chapter mostly focused on privacy, we now look at the implications of using data privacy on the integrity of data, with respect to the extent to which they are verifiable or can be manipulated.

Verifiability

Verifiability of ledger state is one of the main pillars of blockchain, albeit for eliminating trust or for auditability purposes. In practice, it means that someone is able to derive the current ledger state by applying all transactions starting with an empty state. For both frameworks, simply executing all transactions would now obviously be insufficient, as private data is never recorded on the ledger's blockchain. Yet, both frameworks record a trace of the private transaction on the public ledger, which could be used to verify private state. In Fabric, this trace is actually a hash of the private state, which by definition satisfies the endorsement policy for it to be recorded on the main ledger. Thus, to prove the validity of some private state, the (unhashed) private state can be revealed. Then, a hash can be computed over that private state and compared with the hash recorded on the main ledger. This is unfortunately not possible with Besu, as the hash on the public ledger does not testify of state, but only that a certain private transaction was created. Depending on the smart contract, revealing the private transaction could be insufficient to derive the state that results from the execution, since the transaction may depend on its previous state. For instance, to transfer an asset from A to B, the smart contract may only modify state if A actually owned the asset. Because of this dependency on prior states, one may have to execute *all* private transactions of the private group before one can verify if some "supposed" private state is provably correct.

Prevention of tampering

Another important aspect of data integrity involves the extent to which a member can tamper with the data for its own benefit. Since transactions in Besu have to be executed on each individual node, just like with public transactions, it is impossible to force an honest node to make illegal changes to its ledger state. Yet, it would be possible of course for Byzantine nodes to change their own local state, but an honest node would never accept any new blocks created by such

Byzantine nodes as the state's root hash would differ. Although it is possible for all members of a private group to make the same illegal changes, especially as verifying their ledger state is a rather complicated process as we have seen before. But since nodes may only interact with members in that same private group, without interacting with other groups, it would be rather meaningless if all private group members tamper with their state. Namely, they could have simply deployed another smart contract instead, to make those changes in a *legal* (untampered) manner (albeit in a new SCA's namespace).

For public transactions, we have seen in Section 3.1.9 that changes made to state are only as secure as the endorsement policy defined. Recall that as long as an endorsement policy is satisfied, any custom readset and writeset could be crafted that was not generated by chaincode execution. For private transactions, the very same mechanism is in place, since each PDC defines its own endorsement policy. Yet, at the same time, we have seen in Section 6.1.1 that the documentation actually encourages the use of smaller PDCs rather than larger PDCs. Even implicit PDCs, which consist of a single organization, has "a(n) [...] endorsement policy of the matching organization"[8]. When the private data is irrelevant for non-members of the PDC, such when it involves a simple bid or purchasing price, tampering with the private data is not that big of an issue. However, when another organization that is not a member of the PDC needs to trust the private data is authentic, tampering is problematic, as the following examples show.

Consider the supply chain for ethically-sourced mining resources as introduced in Section 6.1.1. In a permissioned blockchain with high competition, we saw an organization might want to keep information such as the price and quantity of a shipment's resources private. If the PDC-members would tamper with the price, the consequences are limited since this information does not affect other organizations down the chain. However, when the quantity of a shipment is changed, such as when it is mixed with unethically sourced goods, it becomes a much larger problem (for other organizations further along the chain). Similarly, when dealing with second-hand vehicles, storing the odometer in a single-organization PDC, means that the organization can craft a writeset to update the asset and distribute it as an endorsed transaction. Since it is by default the only organization required for endorsement, and others only see that the asset's hash is updated, they assume that the change is valid. By revealing the asset to some or all organizations at a later point in time, means they all see a vehicle with a low odometer while matching the public ledger's hash, not knowing the data is actually illegal.

These examples illustrate that even when PDCs are used, data integrity should not be taken as granted and one should think about the importance of the data stored in each PDC. Simple solutions could be to (1) record information that should provably be valid on the main ledger for greater transparency, (2) include a notary organization in PDCs with provably valid data, (3) include more (actively participating) network participants in the PDCs and their endorsement policies. Admittedly, all of these solutions result in less privacy. Another solution could be to enforce that once an asset (and its hash) has been created, no participant may be allowed to modify it. This allows an asset to be provably tracked from the source. Perhaps, future work could be to further research unmodifiable keys, that may only be transferred between PDCs, in combination with the enforcement that only a subset of organizations may create new keys on the ledger (i.e. sources/manufacturers). By enforcing this in the VSCC, an honest node would only accept some organizations to write a key, after which it may only be transferred, resulting in integrity of the data over time.

[8]https://hyperledger-fabric.readthedocs.io/en/release-2.2/private-data-arch.html

Overview of Private Data Integrity

Private Data Integrity		
Concept or Feature	**Fabric**	**Besu**
Verifiability of Private Data	By disclosing the private data containing private state, a hash can be computed and compared with a hash already stored on the public ledger.	To prove a private group's state, all private transactions (including private contract creation) need to be disclosed and executed from the moment the private group was created.
Preventing Tampering	Endorsement policy testifies for data creation. If too weak, data of PDC can be manipulated for organization's own benefit. Requires strong considerations on the composition of PDCs and the data stored in them.	All private group members need to change key-value state, so effectively meaningless for honest nodes. Any private group composition is harmless for honest nodes.

Table 6.2 Overview of Private Data Integrity

6.3.3 Implications for Data Availability

Transaction Creation and Node Availability

In Fabric, creating a private transaction is no different than creating a public transaction, besides having to satisfy the PDC's endorsement policy. This means that all nodes not required to satisfy the endorsement policy are allowed to be unavailable, while still being able to create, endorse, order and validate a new transaction. Besu is fundamentally different in terms of required node availability. In Besu, it is no longer allowed to have a quorum of nodes online, i.e. two-thirds as we have seen with IBFT 2.0, but Besu requires "*all* private transaction participants (to) be online"[9]. In practice, this requirement is enforced on Orion nodes that only accept a `send`-request if it could disseminate the private transaction to all other Orion nodes in the private group, as we have seen in Section 6.2.1.

Because a private group in Besu is unable to interact with other groups or write to the main ledger, it means a business application is deployed in complete isolation within the private group. Since no interaction from outside the private group is possible, all nodes with an interest in the deployed smart contract have to be explicitly defined as a member of the private group. In other words, private groups are bound to be large and groups of only 1 or 2 participants are rather meaningless. It should be apparent how requiring all private group nodes to be online could be a very serious issue.

Retrieving Missing Private Data

To investigate the discrepancy between Fabric's loose and Besu's strict availability requirement, we dived in Besu's and Orion's source code. While processing a block, the Besu node only checks if Orion has the value *already*. If the Orion node does not have the private transaction identified by the private marker, it will not attempt to see if other members of the same private group have that private transaction. Moreover, functionality to *pull* private data from other Orion nodes does not even exist! That also explains why a `send`-operation only succeeds if it could be pushed to *all* other group members (instead of two-thirds, for example). Otherwise, if just one node did not receive the information, it has no way whatsoever to retrieve this data. At the same time, this explains why private groups in Besu are static and cannot be modified after creation, namely, they have no way to retrieve the private state up to that point. Since Fabric nodes *are* able to pull data from other members of the PDC as part of the Gossip protocol, it does not matter if a large amount of nodes do not have this information yet.

Permanent Data Loss

The fact that an Orion node is unable to retrieve private transactions from other nodes on request has another immediate and serious consequence: A member that loses its private data (after it has received and acknowledged it through `send`) is

[9] https://besu.hyperledger.org/en/stable/HowTo/Use-Privacy/Run-Orion-With-Besu/

never able to retrieve this data (from others) again. This could have various reasons, such as human error or a hard drive failing, for instance. Although Orion could be configured to store private transactions in a Fault-tolerant database such as Oracle Database or PostgreSQL, this is still not optimal and truly decentralized as these instances are managed by the same entity. Although one might argue this could be the member's own responsibility, losing this data due to a mistake also impacts all other members in that private group, as they can no longer transact with that node. Therefore, it is in the entire group's best interest for all members to have all private data. In essence, a Besu member that loses (some) of its private data, can no longer participate in the private group (at least using the latest implementation of Besu/Orion).

Fabric does not suffer from this issue due to its ability to pull private data from other Fabric nodes in the same PDC. Whether data pulling is supported or not, if *all* nodes of a private group lose their private data, this data is also lost forever. For Fabric, this mechanic can actually be configured to happen on purpose by setting the `blockToLive` property. Then, a certain amount of blocks after private data was written, that data is discarded by all nodes. Note that this does *not* result in inconsistent private state, as all PDC members drop the same information at the same block height. Alternatively, when certain private data is very important and *may not* get lost in any case, only a pulling technique may be insufficient for the required level of data fault tolerance: E.g., when the PDC is very small or consists of a single organization. Then, the Fabric node could be configured to use CouchDB in cluster mode, meaning each Fabric node stores its (private) data in a fault-tolerant manner.

Final remarks on Besu's data availability Section 6.2.3 revealed that the Besu team is currently experimenting with flexible privacy groups. Although incomplete, they were able to add and remove members from private groups, while their Orion nodes still cannot pull private data from each other. Instead, they implemented the following solution: When a member is added to a private group, a Besu node that is already a member of the private group retrieves all of his private data from Orion (for that private data collection), and pushes it to the newly added member. While this solves the problem that private groups cannot be changed after creation, it does not relax the strict availability constraint, and it does not solve the risk of permanent data loss by a private group member. A more sensible approach would be to allow Orion nodes to make pull attempts to other Orion nodes of the same group when it receives a request for a key it does not have locally, similarly to Fabric's implementation.

Further research revealed that the Besu team might be reconsidering its integration with Orion. ConsenSys, the main contributor behind Besu and Orion, has been developing another Ethereum-based permissioned blockchain framework called GoQuorum[10]. Like in Besu, GoQuorum also supports private groups and stores private data in an external component, called Tessera[11]. Based on some of the recent GitHub issues[12], integration of Besu with Tessera is currently being tested. A page on Tessera's documentation claiming "Tessera provides out-of-the-box support for Hyperledger Besu"[13] testifies of this as well, although this feature is not released in an official version (yet) at the time of writing. While private groups in Tessera are (still) unmodifiable like in Orion, it *does* allow a member of a private group to recover lost data by pulling it from other Tessera nodes.

[10] https://docs.goquorum.consensys.net/en/stable/
[11] https://docs.tessera.consensys.net/
[12] https://github.com/hyperledger/besu/issues/2045
[13] https://docs.tessera.consensys.net/en/latest/HowTo/Configure/Orion-Mode/

Overview of Private Data Availability

Private Data Availability		
Concept or Feature	**Fabric**	**Besu**
Individual, permanent data loss	No, if another member of the PDC still has the private data, the Fabric node that misses the data will fetch this information from another node.	Yes, when a single private group member loses its private data, it cannot fetch this from others and the peer is lost.
Collective, permanent data loss	When all members of the private group or collection lose their private data, the private data is lost forever as the private data is only stored by the private group/collection members and not on the public ledger.	
Coordinated, collective private data deletion	Supported, by defining a maximum amount of blocks to store the private data.	Unsupported.
Fault-tolerant data store	Supported: CouchDB.	Supported: Oracle Database and PostgreSQL.

Table 6.3 Overview of Private Data Availability

Chapter 7

Benchmarking Fabric and Besu

Most chapters up until now focused on the theoretical aspects of the frameworks. This chapter, on the other hand, has a more practical goal, namely to investigate how both frameworks perform in practice. Section 7.1 gives an overview of the various performance metrics that will be measured during the benchmarks, and Section 7.2 describes the methodology of executing benchmarks, as well as the smart contract used. The benchmarks are conducted as a series of experiments, as reported in Section 7.3. Lastly, our results are put in perspective with the literature in Section 7.4.

7.1 Performance Metrics

First of all, we are interested in the **throughput** of both frameworks using different configurations. We consider throughput as the number of transactions per second. Since it is likely that throughput continuously fluctuates over time and we are mostly interested in the throughput over a longer timeframe, i.e. small perturbations are mostly noise, the throughput is computed over the entire timeframe of each experiment.

$$N = \text{Total number of transactions} \tag{7.1}$$

$$t_{cr,i} = \text{Creation time of transaction } i \tag{7.2}$$

$$t_{co,i} = \text{Commit time of transaction } i \tag{7.3}$$

$$Throughput = \frac{N}{\max\limits_{t_{co,1}\cdots t_{co,N}} - \min\limits_{t_{cr,1}\cdots t_{cr,N}}} \tag{7.4}$$

Although throughput is often considered as one of the most important metrics for blockchain performance, **latency** could be just as important as well: some use cases might only need a throughput of 5 transactions per second while requiring a very low latency of 1 second, whereas other use cases might not care if it takes up to five minutes to commit a transaction, as long as the network can handle hundreds of transactions per second. We define latency as the time it takes between the creation and commit of a transaction. In contrary to throughput, *just* the mean latency does not give enough information. Recall from Section 4.2.4 that in Besu, different transactions will have a different priority. Technically, it could happen that most transactions always have a low latency, while the latency of some other transactions are much lower, resulting in starvation. Thus, the minimum and maximum latency is also tracked, just in case we find some interesting anomalies in the results of our benchmarks.

$$Latency_i = t_{co,i} - t_{cr,i} \tag{7.5}$$

$$AvgLatency = \frac{\sum_{i=1}^{N} Latency_i}{N} \tag{7.6}$$

$$MinLatency = \min_{Latency_1 \ldots Latency_N} \tag{7.7}$$

$$MaxLatency = \max_{Latency_1 \ldots Latency_N} \tag{7.8}$$

$$ErrorRate = \frac{FailedTxs}{FailedTxs + SucessTxs} * 100 \tag{7.9}$$

We keep track of the percentage of failed transactions through the **Error Rate**, which is the percentage of failed transactions. Since failed transactions are never included in blocks in Besu (otherwise the entire block would be invalid), and never applied to the ledger of a correct Fabric peer, failed transactions are not included in the latency and throughput.

The Throughput and Latency are *network* performance metrics. Namely, they are measured by looking at the entire collection of nodes, rather than at individual nodes. In contrast, we could also take individual container metrics into account related to **Resource Consumption**. Four types of resource consumption metrics are measured. First, we have average CPU usage and average memory consumption in MB. This could be relevant for IoT devices, for instance, that should be as cheap as possible while using as few resources as possible. To obtain reliable values, CPU and memory should periodically be observed (i.e. every 5 seconds) from the runtime environment and aggregated after the experiment has finished. Since this definition is rather trivial, their formulas are omitted. In addition, we look at the total amount of network bandwidth used by each node, both for incoming and outgoing traffic, by taking the difference at the start of the benchmark and the end.

7.2 Experimental Set-up

Benchmarking is conducted with the help of Hyperledger Caliper[1], which is a tool that aims to become a standard for benchmarking a wide variety of blockchain implementations. By using this tool, we can ensure that benchmarks are conducted across multiple frameworks by following a similar flow, i.e. to reduce the effects of the benchmarking tool. The test networks are deployed in Docker on a consumer-grade desktop computer with the specifications depicted in Table 7.1. Since the benchmark is conducted on a single device including the load generation, the reader should be aware that the hardware is shared. However, since experiments are conducted in a controlled environment, all non-essential applications are closed during benchmarks to minimize their impact on the results. Four out of the twelve available threads are reserved for load generation.

Operating System	Ubuntu 20.04
Virtualization Software	Docker 19.03.13
Central Processing Unit	AMD Ryzen 5 5600x
Dynamic Random-access Memory	32GB 3600MHz
Storage	Samsung 970 Evo Plus 2TB (M.2 SSD)

Table 7.1 Overview of Benchmark System specifications

To further minimize the amount of random noise in our results, each benchmark is preceded by a warm-up period of `10.000` transactions. This ensures that the network is initialized at full capacity before taking any measurements. After this warm-up period, the benchmark itself is immediately started over which the various performance metrics are computed. Each benchmark lasts `50.000` transactions. There is no waiting time or cooldown period in between the warm-up period and actual benchmark.

[1] https://hyperledger.github.io/caliper/

Smart contract design is vital to the comparability of benchmarking both frameworks. Therefore, the smart contracts are simple, but representative of many production use cases. Namely, a stripped implementation for Non-Fungible Tokens (NFTs) [32] is written. These encompass the idea that an asset can be created on the ledger that is unique with its own properties. As such, this asset cannot be interchanged with any other asset in the same smart contract. This could resemble supply chain use cases such as a shipment with unique transport data, a certificate of authenticity of jewelry, ownership of an object such as a vehicle, et cetera. A `CreateAsset`-function is called before the warm-up period to initialize the ledger with `500` assets. This is a simple no-reads-single-write operation. During the actual benchmark, each transaction executes a `TransferAsset`-function. This is a single-read-single-write operation, as a transfer is only allowed when the invoker is defined as the owner of the asset, i.e. the EOA in Besu or organization in Fabric that created the asset.

An important aspect of the transferring benchmark, is how many assets there are and how often the same asset is being transferred. Transferring an asset only once is unrealistic. As an IBM colleague mentioned about the supply chain industry, assets change ownership all the time. Even on a single cargo ship, an asset's ownership could change every couple of minutes. To simulate this behaviour, every benchmark includes a total number of assets set to `500`. While generating and submitting transactions, the transaction generators loop over these assets from start to end. When an iteration has transferred all assets, the benchmark continues with transferring the first asset of the list of 500. Thus, for a benchmark that lasts 50.000 transactions (excluding the warm-up period), every asset is transferred $\frac{50000}{500} = 100$ times. This aspect is especially important for Fabric's MVCC. Namely, the less assets there are and the higher the network load, the higher the chance of two transactions transferring the same asset, resulting in a failed transaction. As Besu executes transactions after submitting and ordering them, using an account-based nonce system for preventing "double-spends", the chosen asset count is not expected to influence Besu's results.

Changes required to Caliper

Although Hyperledger Caliper quickly proved to be very flexible and powerful for getting started on designing benchmarks, it was not a perfect solution for our use case and some changes had to be made. First of all, we encountered a bug involving the resource monitoring. Our benchmarks were set-up as a three-round execution: (1) asset creation, (2) warm-up period, and (3) the actual benchmark. Resource monitoring (for Docker) was actually only implemented for the very first round, in our case, the asset creation. Ideally, resources should be monitored for each of the three rounds separately. After some debugging, we discovered that the containers to monitor would be set to an empty array, as opposed to an expected `null`. The bugfix was committed to the Caliper project[2].

Secondly, Caliper lacked a feature we would need, namely the option to submit transactions from multiple Ethereum EOAs. Normally, one could only submit transactions from the same account for the entire benchmark. Since Ethereum works with account-based nonces, this would mean that transactions can always only be executed in a single thread (as the nonce should atomically increase with every request). We added a feature that allows users to specify a `sender.nonce` and `send.privateKey` to each individual transaction, allowing transactions to be sent in parallel from different accounts. This new feature was also proposed to the Caliper project[3].

Thirdly, Caliper implemented a waiting time of 5 seconds between each round by default. To truly ensure that the warm-up round is immediately followed by the benchmark, this 5-second waiting time was removed. As this involves more of an implementation-specific *hack*, this was not committed to Caliper for obvious reasons.

Lastly, using Caliper with private transactions is undocumented. After talking to the Caliper developers for Fabric and Besu (both framework connectors are maintained by different people), they gave us some guidelines for submitting private transactions. Eventually, we got this working by setting some additional fields to each transaction (move arguments to `transient` for Fabric, and `privacyGroup` for Besu). Unfortunately, however, private transactions in Besu in combination

[2]https://github.com/hyperledger/caliper/pull/1135
[3]https://github.com/hyperledger/caliper/pull/1138

with Orion involves a memory leak. Every time a Besu node receives a private transaction, it sends the transaction to Orion, which uses *Apache Toweni* for PKI communication. Every time a `PublicKey` object is instantiated, it results in a call delegated to `LibSodium`, a program written in C, which calls `malloc` for every newly created object. During our benchmarks, this resulted in memory leaks, causing the Orion nodes to crash after about 400 transactions. Since not a single transaction can be successfully submitted after this point[4], benchmarking Besu's private data with Orion is infeasible at the time of writing.

7.2.1 Variables and Base Configuration

The performance metrics of both frameworks rely on a lot of variables, too much to test every possible combination. Therefore, a base configuration is constructed for each of the two frameworks. Consequently, these configurations are used for all of the experiments, unless stated otherwise. The base configurations of Fabric and Besu frameworks are given in Table 7.2 and Table 7.3, respectively.

Contract Visibility	Public
Number of Organizations	3
Number of Peers per organization	1
Number of Ordering Service Nodes	2
Block Time	2 seconds (default)
Block Limit	500 transactions, 2MB preferred, 10MB absolute maximum
Consensus Protocol	Raft

Table 7.2 Fabric Base Configuration

Contract Visibility	Public
Number of Peers	3
Block Time	2 seconds (default)
Block Gas Limit	12,500,00 gas (default)
Transaction Pool Limit	4096 transactions (default)
Consensus Protocol	IBFT 2.0

Table 7.3 Besu Base Configuration

7.3 Experiments

7.3.1 Baseline performance

For our first experiment, we are interested in investigating the performance of Fabric and Besu using our base configurations. We start by a network load of 10 TPS. We keep doubling the network load until we start seeing a significant drop in throughput, increase in latency, or increase in failed transactions. In between the network loads where such changes occur, additional benchmarks are conducted to obtain a more fine-grained view.

Performance and Error Rate

Figure 7.1 shows the throughput and error rate for both Fabric and Besu. On the x-axis, we see varying network loads. The left y-axis shows the throughput of the network in TPS, corresponding with the solid lines. The right y-axis shows the error rate in percentages, which are illustrated by the dashed lines.

We start by looking at the throughput of the network. It immediately becomes clear that the throughput of a Fabric network grows linearly as the load increases, until a load of about 700 TPS. No data points are shown higher than this, because our

[4] https://github.com/ConsenSys/orion/issues/425

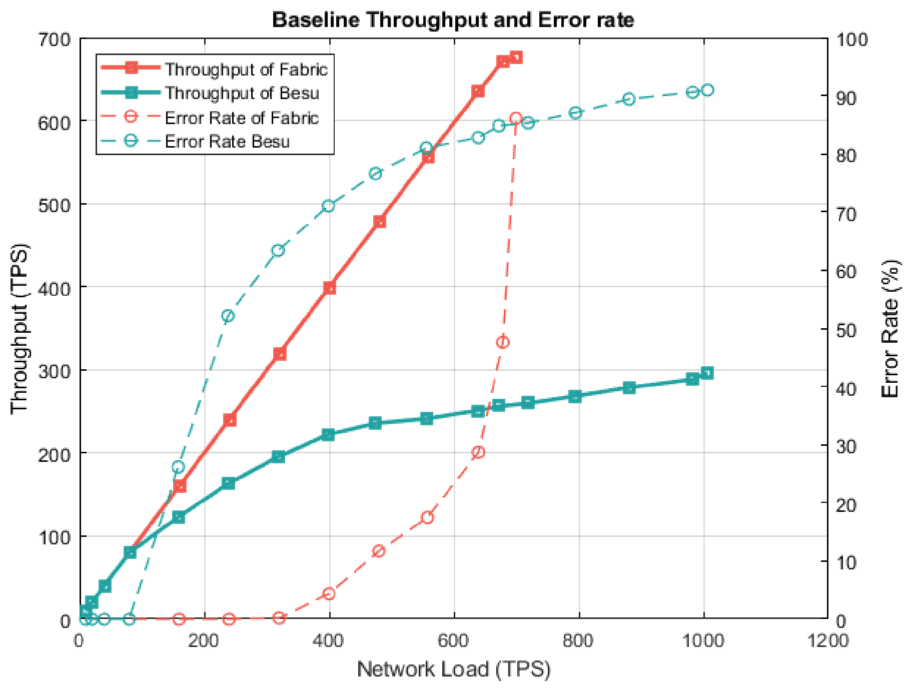


Fig. 7.1 Baseline Throughput and Error Rate

set-up cannot generate transactions faster than this rate. Our load generator runs on the same machine as the Fabric nodes. To prevent too many context switches and resource contention caused by having too many threads, our load generator is limited to 4 threads. These four threads are only able to generate and send up to about 700 TPS. This shows that the network likely has not reached its bottleneck for the amount of load we were able to generate. To push the network further, all nodes and the load generator would have to run on separate servers, which is not in the scope of this thesis. Besu's throughput looks completely different. Up until a load of about 80 TPS, the throughput exactly matches the network load. After a load of 80 TPS, however, we see the throughput starts to drop significantly. The shape of Besu's throughput curve resembles an inverse logarithmic shape, showing the network cannot process the transactions as fast as they are generated, i.e. the Besu network is approaching its maximum potential for the current set-up. Moreover, Besu's throughput keeps slowly increasing, even after 700 TPS, at which our load generator for Fabric stagnated. This data confirms our findings in Section 4.3.1 that Fabric's transaction creation involves much more responsibilities, causing it to be more computationally expensive than Besu, which is mostly fire-and-forget.

The error rate of Fabric and Besu are very dissimilar as well. Our Fabric network only starts reporting errors at a network load of more than 320 TPS. The errors caused by Fabric are thrown by the MVCC, meaning that the input of a transaction that is ordered in a block has been overwritten by another transaction in the meanwhile. Although this is a direct consequence of our benchmark design, i.e. every asset is transferred 100 times, it is also a consequence of Fabric's design decision to execute transactions before ordering. The higher the network load, the higher the chance a MVCC-error occurs, explaining the exponential shape of the error curve. These errors, on the other hand, do not occur in Besu because a transaction does not include the versioned readset and writeset. Instead, Besu's errors are caused by a different aspect: Besu's transaction pool is limited to 4096 transactions by default. If this pool's capacity is exceeded, transactions are dropped. Since transactions are generated with monotonically-increasing nonces, and a transaction with a higher nonce waits for its prior nonce to be executed first, a transaction with nonce `n` that is dropped causes all transactions with nonce `n+i` to fail as well. If the network load is increased, the point at which transactions (and all transactions of the same account after that) fail occurs earlier, resulting in a slightly higher error rate (since less transactions succeed), explaining the inverse logarithmic curve of Besu's error rate.

Latency and Error Rate

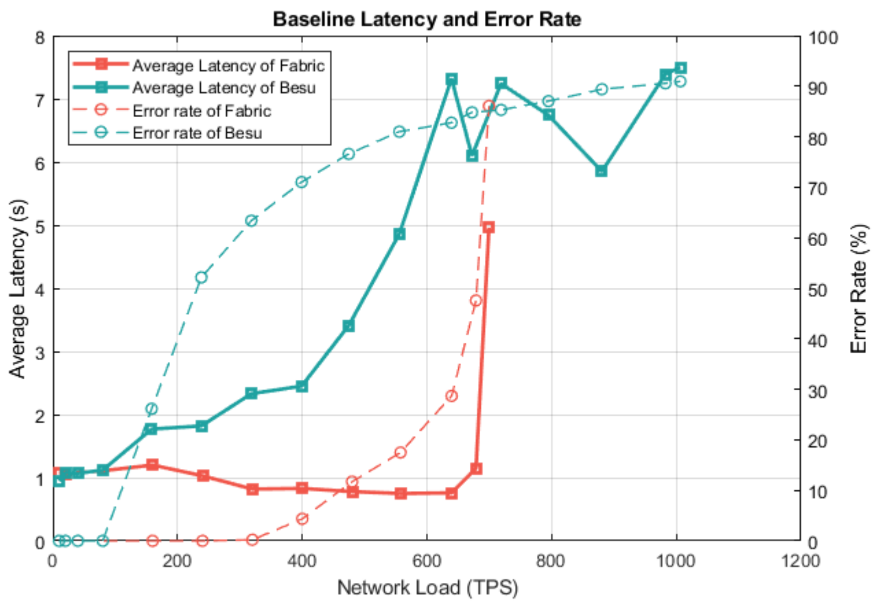


Fig. 7.2 Baseline Latency and Error Rate

In Figure 7.2, we see the average latency of successful transactions in seconds. We observe that the average latency of Fabric and Besu both start slightly higher than 1 second. A latency of about 1 second under full utilization is actually what we expect: Since both frameworks have a target block time of about 2 seconds, the first transactions in a block have to wait for about 2 seconds, whereas the last transactions in a block wait close to 0 seconds. Assuming transactions are added to blocks evenly distributed over time, transactions would sit for 1 second on average in a block. The actual average latency is slightly higher, presumably due to the overhead of executing/appending and verifying the blocks.

By observing Fabric's Latency graph, we see that the latency first starts dropping below 1, as the network load gets above 240. In Fabric's transaction flow, we saw that a block is cut when its maximum capacity is reached, even if its block time is not exceeded. This is exactly what we see in the graph: blocks are created faster than the configured block time to adapt to the network load, explaining the latency drop. Between a network load of 680 and 700 TPS, Fabric's latency shoots up to 5 seconds and above. This sudden increase is caused by the fact that the pending transactions are starting to create a backlog, faster than they can be cut in blocks or processed by the peers. In other words, the buffer of pending transactions is close to 0 for a network load less than 680, but starts to stack up from there.

Besu's latency chart behaves slightly differently. Latency never drops below 1 at the start as the load increases, because the Besu network does not create blocks at a faster pace than the block time, even if there is a backlog, as we have seen in Besu's transaction flow. We see that Besu's average latency stays fairly consistent slightly above 1 seconds up until a network load of 80 TPS. After that, the latency starts to go up. Especially in this figure, the relationship between latency and error rate is important. Namely, as soon as the error rate starts to increase, meaning the transaction pool is full, the latency starts to go up for successful transactions. In other words, as soon as we cannot keep the delta of the transaction pool inflow at 0, and the transaction pool starts to fill up, we have a backlog of transactions. This backlog translates to an increase of average latency as well. When the network load is above 640 TPS, the latency starts to fluctuate a lot. A sound explanation for this effect cannot be given, besides the fact that around 85+ percent of the transactions fail, and the latency is measured over a too small sample of successful transactions.

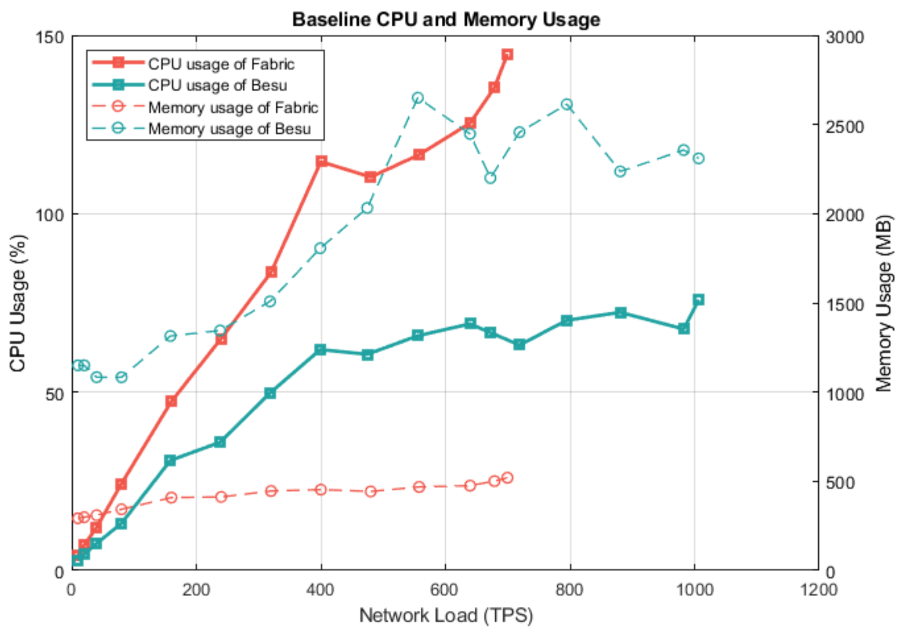


Fig. 7.3 Baseline Resource Consumption

Resource Consumption

The average resource consumption of both frameworks is illustrated by Figure 7.3. Fabric's CPU usage is much higher than Besu's. After a network load of 320 TPS, we even see Fabric's CPU usage exceed 100%. This is not a mistake, but simply means that a Fabric peer completely utilizes more than 1 thread on average. Besu's CPU usage seems to approach 60% up until 400 TPS, after which is stays between 60% and 75%. At first, this graph could be confusing with respect to our earlier findings: if we cannot push the Fabric network to its full potential, but we can on Besu, would we not expect Besu's CPU usage to be higher? The answer is clear if we look back at Figure 7.1: the shapes of the curves are resembling of the network's throughput. Since Besu's CPU utilization is much lower than Fabric's, it seems that the CPU is likely not Besu's bottleneck.

By looking at Besu and Fabric's memory, we see a clear difference. Besu uses an average memory of at least 1000 MB, which increases with the network load up until 2650 MB. Fabric's memory usage, on the other hand, is a lot less. Although Fabric's memory usage also grows with the network load from 293 MB to 524 MB, its memory usage during a network load of 700 TPS is still far below Besu's memory usage at 10 TPS.

Network Traffic

Figure 7.4 illustrates the average network Traffic In and Out for each Besu and Fabric node for our baseline network loads. The y-axis of this graph is set to logarithmic scale for readability. Starting with Fabric's network traffic, we see its ingress and egress traffic follow a similar curve: For a network load of 10 TPS, 386 MB of input data is handled on average by each Fabric node. Each node sends on average 217 MB. These two values are not equal, because some data is sent from and received by the load generator and the OSNs. As the network load increases, we observe a trend that causes both ingress and egress traffic to decrease slightly. This (small) drop is to be expected: Since the amount of transactions is kept constant, but network load increases, benchmarks are finished quicker for larger network loads, meaning less maintenance requests (such as heartbeats) are made.

Besu's network traffic, on the other hand, shows some very different patterns. Let's start by looking at Besu's ingress traffic. Under 80 TPS, when there are no failing transactions yet, each node has on average about 55 MBs of incoming traffic,

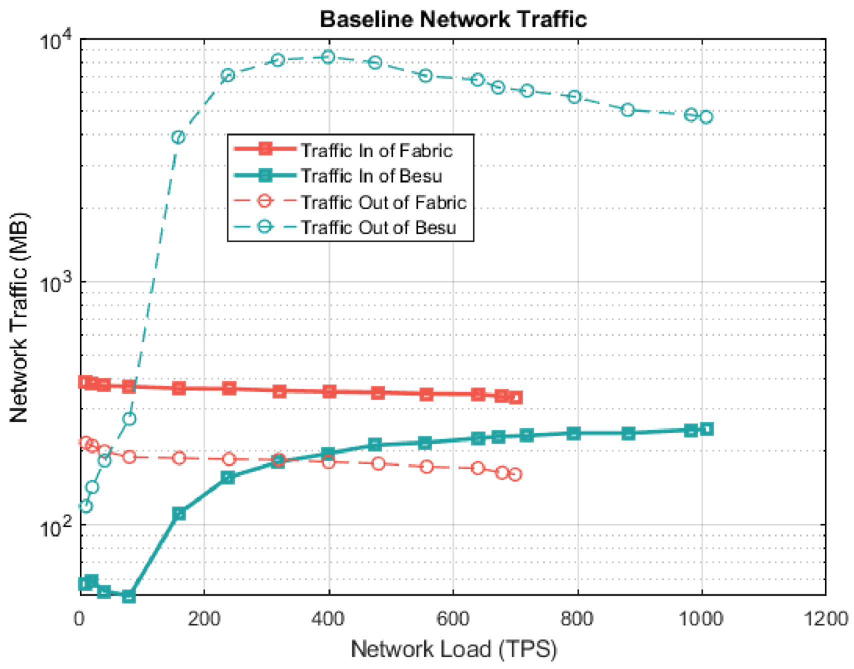


Fig. 7.4 Baseline Network Traffic

which is significantly less than Fabric's ingress. Moreover, it can be observed that Besu's ingress traffic keeps dropping when the network load increases, most likely due to the same reason as Fabric: communication overhead is less due to the benchmark finishing earlier. Starting at a network load of 160 TPS, both ingress and egress traffic starts to increase drastically. This sudden change of trend can be explained when looking at Figure 7.1. Namely, this is the point at which transactions start to fail. The network usage of the nodes suddenly becomes extremely high, compared to its network usage when no transactions would be failing. For incoming traffic, ingress reaches as high as 238 MB (compared to 60MB before transaction failures), and outgoing traffic even reaches 8.4GB on average per node. In addition, the gap between ingress and egress is very large, indicating that a lot of (egress) traffic is not received by other Besu nodes.

To investigate this odd behaviour, we looked into Besu's source code, to see what would happen when a transaction fails due to the pool's capacity being reached. We found that whenever this occurs[5], the node simply returns a response mentioning that the transaction failed. This response size should not be significantly of different size than a successful response. Moreover, all *observers* of dropped transactions are notified. Since these observers communicate through JSON-RPC, nodes only communicate over DevP2P, and we did not manually subscribe to this endpoint, node-to-node communication was eliminated as a potential cause. Then, we dived back into Caliper's source code, which only makes a simple `web3.eth.sendSignedTransaction()` call to determine if a transaction succeeded or failed[6]. After following this trail to Web3.js, which is one of the most commonly used libraries for submitting transactions programmatically, we found the issue. For every *single* request sent through this client library, a new subscription is created to listen to the block headers of every newly created block[7]. This subscription stays open until the submitted transaction was added to a block successfully, or after it has not observed this transaction in the next `N` new blocks. For our benchmarks, the default value of 50 blocks is used, i.e. it waits at least 50 blocks for the transaction to be included. The problem, however, is that Web3.js creates this subscription for *every* single transaction that is submitted. E.g., in case a 1000 transactions are submitted,

[5]https://github.com/hyperledger/besu/blob/f6d50dcab2813574b996263b99415d4c7e365d0c/ethereum/eth/src/main/java/org/hyperledger/besu/ethereum/eth/transactions/PendingTransactions.java#L266
[6]https://github.com/Viserius/caliper/blob/f431d535a5f8e24580a49fd136bd56c24e7e98d9/packages/caliper-ethereum/lib/ethereum-connector.js#L267
[7]https://github.com/ChainSafe/web3.js/blob/5d027191c5cb7ffbcd44083528bdab19b4e14744/packages/web3-core-method/src/index.js#L557

Web3.js creates a 1000 subscribers, which causes the Besu node to send each new block header 1000 times as well (to the same client application!). For successful transactions, this is not too much of an issue because if it is included in a block quickly, a successful transaction closes the subscription immediately, ensuring not too many subscriptions are open simultaneously. For failed transactions, however, all of these subscriptions remain open for at least 50 blocks. To solve this issue, one would either have to communicate to the Besu node directly without the Web3.js library, which would requiring a lot of custom logic to determine if a transaction is successful, or the Web3.js library would have to be updated to only open a single subscriber at a time, regardless of how many requests are submitted. Although this is a finding of our analysis, solving this issue is outside the scope of this thesis.

7.3.2 Impact of Block Time on performance

Both frameworks define a default block time of 2 seconds. This block time determines how often transactions are cut in a new block. For Fabric, the block time only impacts the block creation if a block has not reached its maximum capacity after this amount of time, resembling a time-to-live. When a block's capacity is reached without the expiry of the block time, Fabric continues on the next block after resetting the timer to 0 again. For Besu, a block time also resembles a time-to-live property, namely, a block is cut when the timer expires. In Besu, however, this block time is also a limit on the creation of blocks: nodes aim to create a block for every block time, regardless of the network load. This means that blocks are created when there are no transactions to include at all, but at the same rate when the network is overloaded: if a block was just created, all nodes sleep until the next block needs to be cut, regardless of the backlog. For this experiment, we conduct benchmarks for a block time of `1`, `2` (default), `4`, and `8` seconds. Although Fabric supports subsecond block times (i.e. 0.5s, 0.05s et cetera), Besu defines block time as an integer value. Since this value is also used to determine various timeouts in a round for IBFT 2.0, a block time of 0 seconds cannot be used in Besu. Therefore, we restrict our benchmark to a value of 1 or higher in favour of direct comparability. We conduct these benchmarks on network loads that are carefully selected to include no failing transactions (80 TPS) for both frameworks, until network loads in which both frameworks have at least some failing transactions (320 TPS).

Throughput for varying block times

Figure 7.5 plots the throughput of each framework for varying block times. The network load is given on the x-axis, whereas the y-axis shows the actual throughput. All throughputs are grouped by their network load, such that the difference in block time can be observed side-by-side. For Fabric, we see that the effects of either halving or doubling the block time is negligible, resulting in a difference of at most 0.2 TPS, which is not significant. This effect can be explained by the same reason given during the discussion of our baseline performance's latency in Section 7.3.1: Even if the block time is very high, blocks will still be created if the block's capacity is reached, thus, not waiting for the block time to expire. For our Besu network, we do see that the block time has a significant impact on the throughput of a network. The faster the block time, the higher the throughput. As discussed earlier, this is likely due to the block time putting a limit on the capacity a network is able to handle.

Latency for varying block times

Figure 7.6 shows the latency for varying block times. These results are also consistent with our earlier findings. Namely, for a low throughput for which Fabric's block are not filled, the block time determines how often blocks are cut, and thus, a lower block time corresponds with lower latency. For higher network loads in Fabric, the impact of block time is much smaller. For Besu, block times are relevant for the latency on all network loads: the smaller the block time, the quicker blocks are created, and the lower the latency.

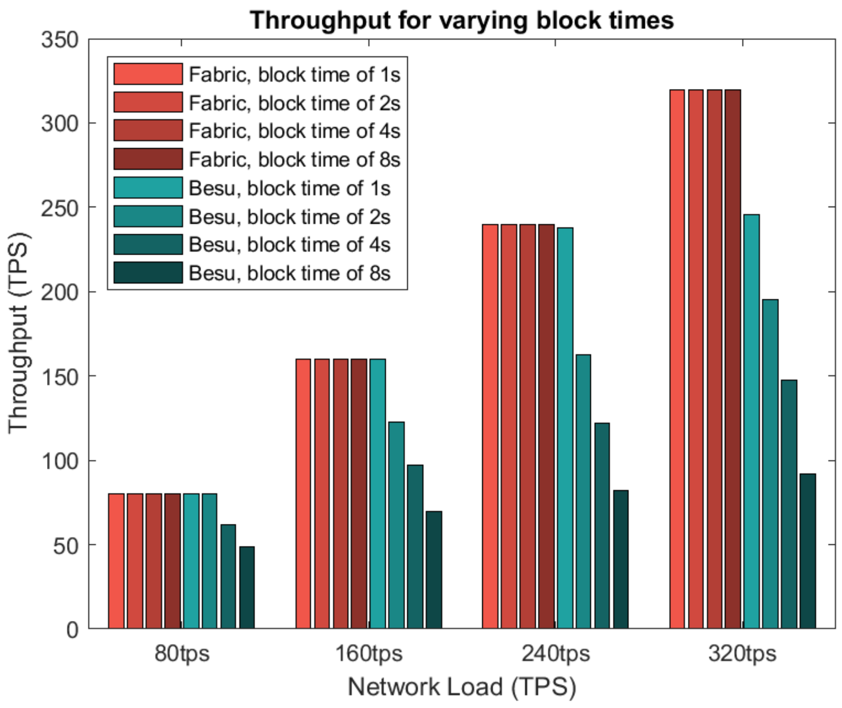


Fig. 7.5 Throughput for varying block times

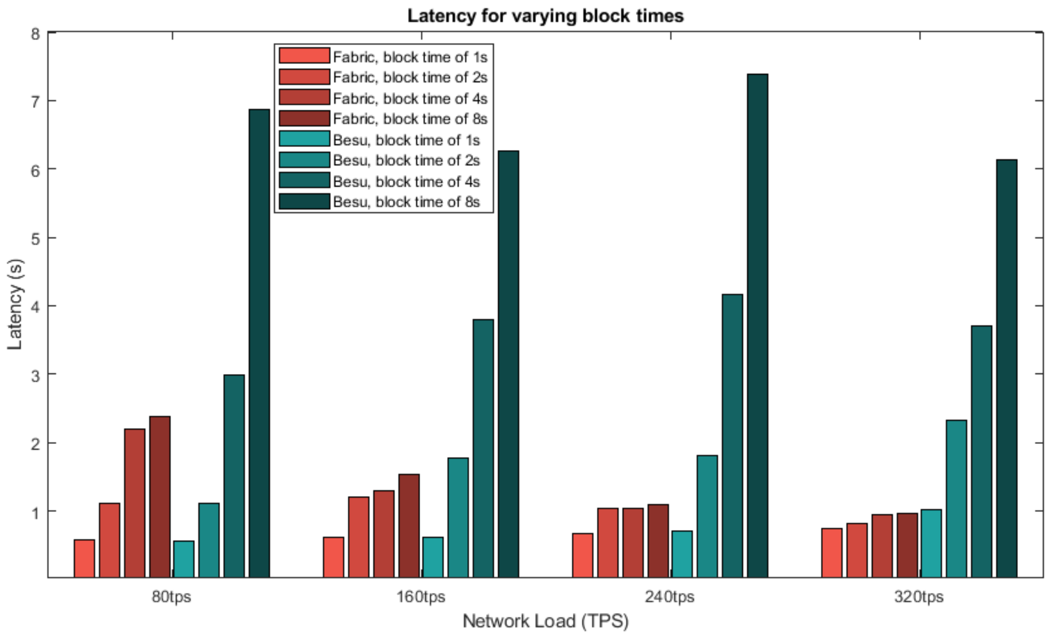


Fig. 7.6 Latency for varying block times

Error Rate for varying block times

Interestingly, reducing the block time could be a strategy to decrease the percentage of failed transactions, as we see in Figure 7.7. Although Fabric's error rate is orders of magnitude smaller than Besu, the same effect occurs, as it has the following error rates for a network load of 320 TPS: 1s block time: 0%, 2s block time: 0.21%, 4s block time: 0.33%, 8s block time: 1.44%. Clearly, the longer that transactions wait to be committed after creation, the higher the chance of a

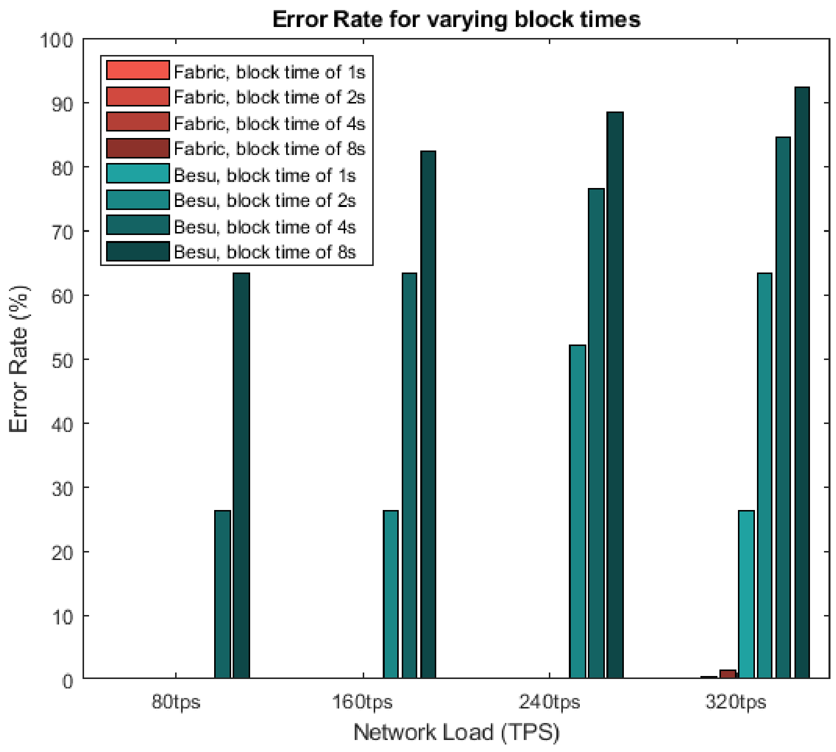


Fig. 7.7 Error Rate for varying block times

conflicting readset in the MVCC. For Besu, a lower error rate for lower block times can be explained by the transaction pool being emptied faster, resulting in a smaller overflow, and thus, a smaller error rate.

7.3.3 Impact of Block Limit on performance

Similarly to block time, every block has a physical limit. For Besu, this block limit is defined in terms of Gas. Recall that Gas maps to a set of weighted operations of the EVM. Some operations are more computationally expensive than others, thus, these operations cost higher Gas. Even in a network in which Gas is free, nodes still keep track of Gas. They simply do not try to charge the submitting account for it in Ether. In Fabric, the block limit is defined using three values: (1) maximum number of transactions, (2) preferred maximum number of bytes, and (3) absolute maximum number of bytes. The difference between (2) and (3) is as follows: the *preferred* maximum of bytes (2) will allow a single transaction to exceed this limit. As soon as the block limit is exceeded, the block is cut. Since transactions can be of different sizes, the *absolute* number of bytes (3) of a block may never be exceeded by a transaction. For instance, when the current-block-to-cut is sitting at 1.5MB, with a preferred capacity of 2MB and absolute capacity of 10MB, any transaction under 8.5MB is still included in the next block. Consequently, if a single transaction exceeds the absolute capacity of a block, it can never be ordered. Whereas if it exceeds the preferred capacity, it can.

As we have seen above, Besu and Fabric define a block limit in different ways: Besu in terms of computations, Fabric in terms of size. Clearly, these measures are not directly comparable with each other. Instead, the default settings that the frameworks ship with are denoted as `DEF` for default, and were listed in Tables 7.2 and 7.3. Now, we can half or double these default block limit values for both frameworks to see their impact on our benchmarks.

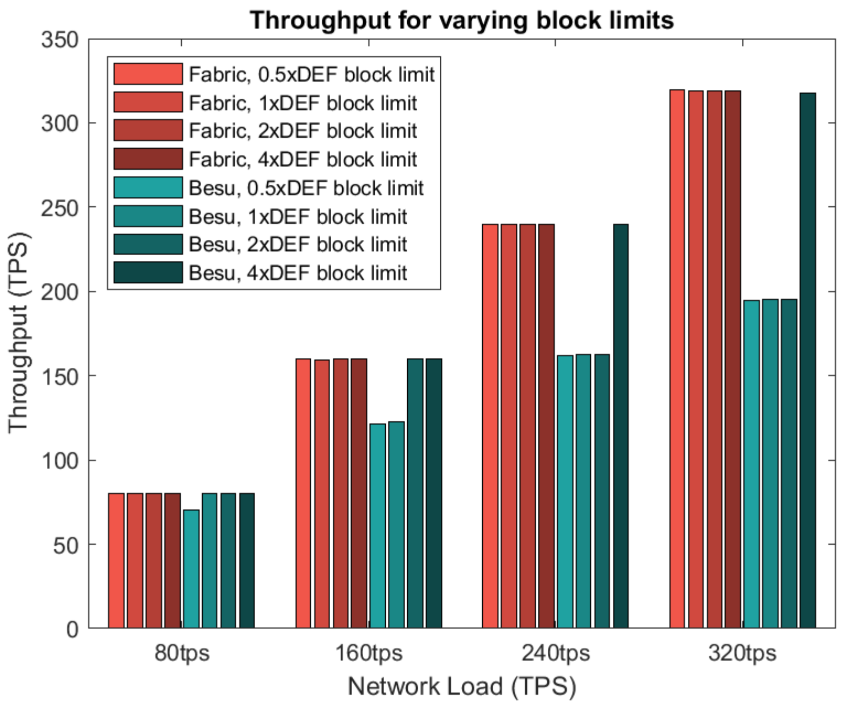


Fig. 7.8 Throughput for varying block limits

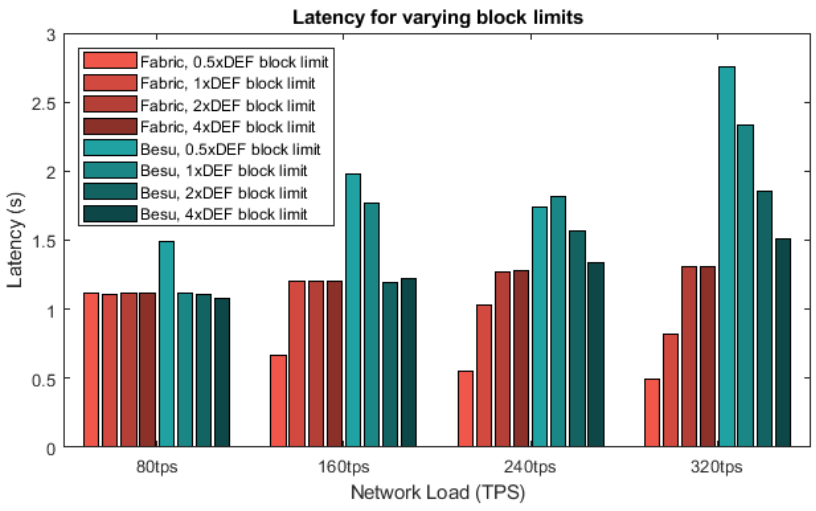


Fig. 7.9 Latency for varying block limits

Throughput for varying block limits

In Figure 7.8, we see that Fabric's block limit has almost no impact on a Fabric network's throughput for the measured network loads. Since the Fabric network already was at full utilization using the default settings, changing the block limit obviously cannot improve the throughput. Yet, from this data, we also observe that changing the block limit to a higher or lower value also does not hurt throughput. For Besu, an interesting observation is to be made. For all tested network loads, increasing the block limit results in a much higher throughput, often as high as the network load itself. Whereas Besu's throughput was previously far under Fabric's throughput, setting its block limit to 4 times the default setting, allows Besu to compete with Fabric for as high as 320 TPS. Since Besu creates blocks at a fixed rate, increasing the block limit allows

Besu nodes to fit way more transactions in each block, preventing the build-up of a backlog. Although throughput could be very high, we should also look at their average transaction latency.

Latency for varying block limits

Figure 7.9 plots the average transaction latency for both frameworks for varying block limits. Although we saw in Figure 7.8 that block limits do not impact the throughput much, a lower block limit can significantly reduce the transaction latency at higher network loads. This is simply the result of blocks being cut more frequently. For Besu, we observe the opposite: the higher the block limit, the lower the latency. As we have discussed before, Besu's blocks are simply not cut more frequently as blocks fill up, and thus, lowering the block limit only means that the backlog fills up faster. This also explains why increasing the block limit in Besu results in lower latency: a transaction is sitting in the transaction pool for a lower amount of time.

Error Rate for varying block limits

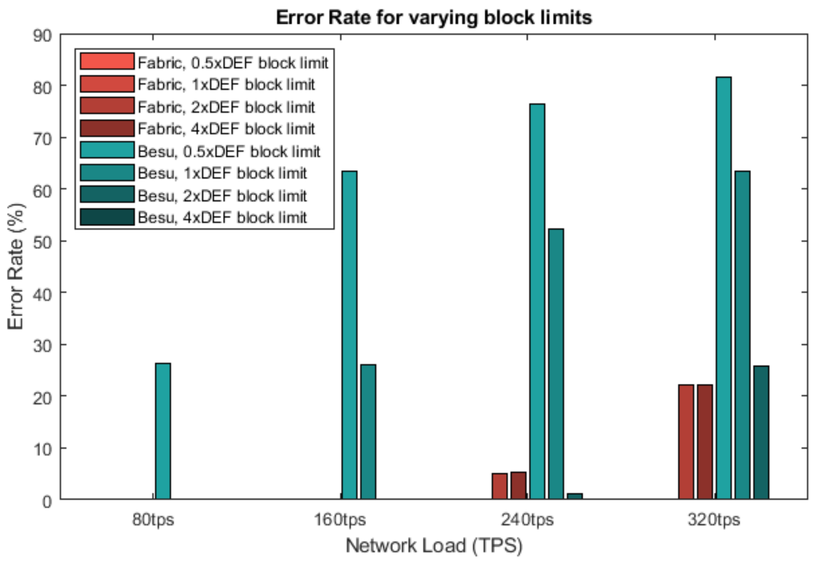


Fig. 7.10 Error Rate for varying block limits

The exact same effect as with latency can be observed for the error rate, for both our frameworks. In Figure 7.10, we see that Fabric's error rate is exactly 0% when the block limit is half the default limit. Since the transaction latency is lower as blocks are cut more frequently, it means transactions are committed faster, resulting in a less likely readset conflict in the MVCC. For Besu, a higher block limit results in less failed transactions, simply because the backlog of the transaction pool is cleared faster. At a block limit of four times the default block limit, Besu even has no failed transactions at all anymore, illustrating the Besu network can be stable without failed transactions at an average latency of 1.3 seconds for a network load of 320 TPS.

7.3.4 Impact of Network Size on performance

Another factor to consider is the Network Size. Since the minimum network size supported by Besu's IBFT 2.0 is 3 peers, we cannot go lower than that. To keep network size of Fabric as comparable to Besu as possible, we define Fabric's network size as number of organizations, each with 1 peer. In this experiment, we conduct benchmarks for a network size of 3, 4 and 5. Since each peer requires its own application and thread, we ideally want to prevent context switching effects in our benchmarks. Therefore, to stay safely below the limit of the amount of 12 threads available on the device performing the benchmarks, we do not go above a network size of 5.

Throughput for varying network sizes

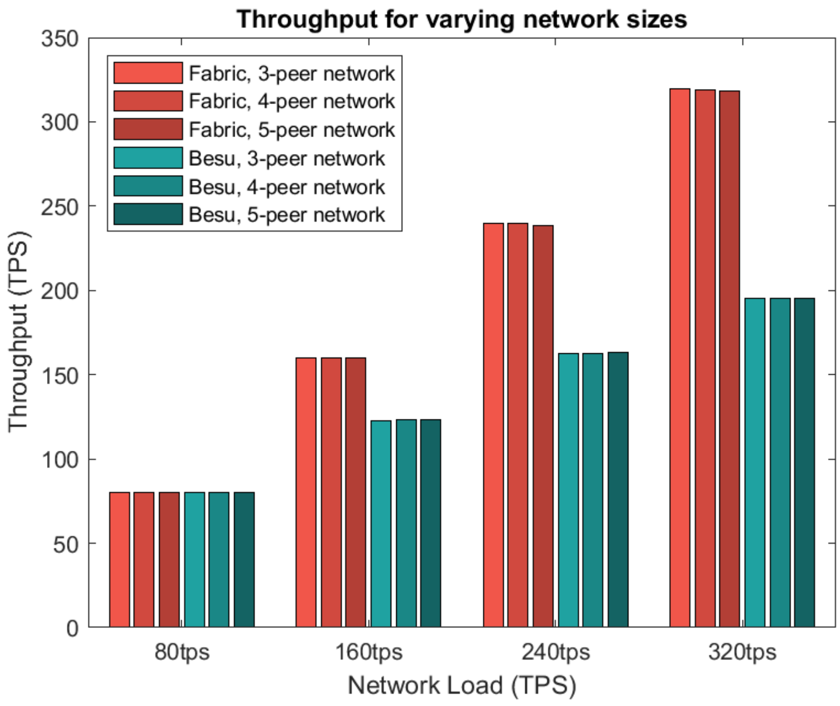


Fig. 7.11 Throughput for varying network sizes

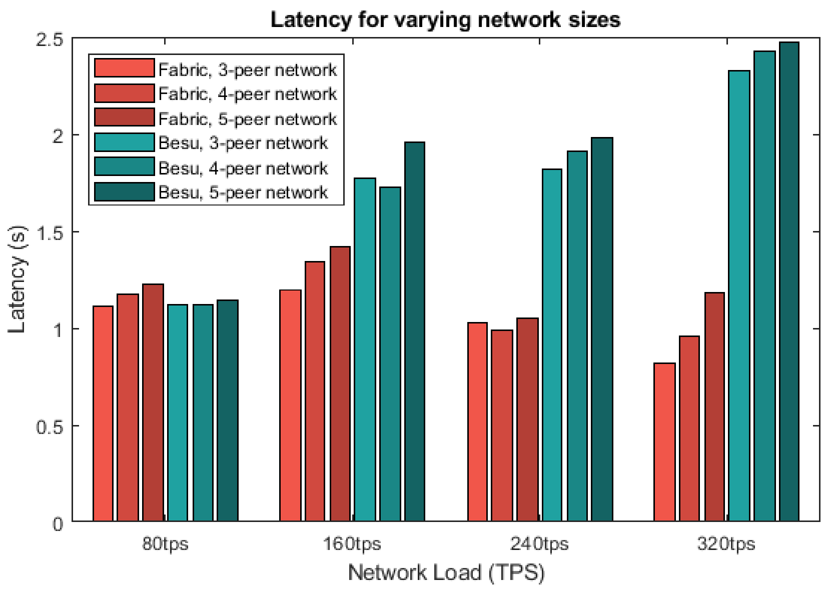


Fig. 7.12 Latency for varying network sizes

In Figure 7.11, we see Fabric and Besu's throughput for varying network sizes. Since the Fabric network is already at its maximum utilization, we expect the throughput not to be higher. However, at the same time we see that adding more nodes to the network does not hurt the throughput, for these network loads, even though our endorsement policy is set to `MAJORITY ENDORSEMENT`. This means that in contrast to a network size of 3 peers, a 4-peer or 5-peer network requires one additional endorsement for every transaction. More interesting is to see the throughput of Besu, considering that this network has reached its bottleneck for our baseline experiments. Since we have seen that Besu's IBFT 2.0 consensus protocol requires two-thirds majority at multiple stages, a network of 3-peers requires consensus among 2 peers, a 4-peer network among 3 peers, and a 5-peer network among 4 peers. Although traditional distributed systems usually achieve higher performance or throughput when adding more hardware resources (i.e. horizontal scalability), it was expected that this is not the case in blockchain systems due to their consensus protocols. Still, it is useful to know that increasing the network size by 66% from 3 to 5 peers does not significantly hurt the network, at least for our set-up.

Latency for varying network sizes

Although we have already seen that throughput is barely effected by the network size, Figure 7.12 shows that Latency *is* somewhat effected. Recall that latency is measured from the time one creates a transaction until it is committed. Generally speaking, this chart tells us that as the network grows, latency also slightly increases. For both frameworks, block distribution could potentially play a role in this, as the one peer that has created a block has to distribute it to either some (through gossip in Fabric) other peers, or all other peers (in Besu, depending on the dynamically formed network edges). Another contributing factor is likely the consensus protocol of both frameworks, since both protocols require more peers to consent before a transaction or block may be committed.

Error Rate for varying network sizes

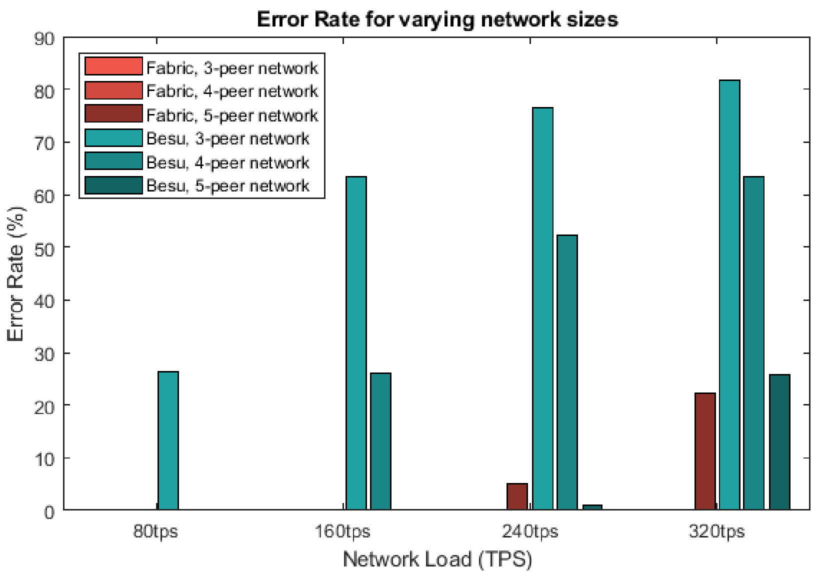


Fig. 7.13 Error Rate for varying network sizes

Although we have seen that the effects of network size are very limited on throughput and latency, Figure 7.13 illustrates that it may have significant consequences on the error rate. In this chart, we see Fabric's error rate for a 3-peer and 4-peer network (close to) 0%. However, when a fifth peer is introduced, the error rate starts to shoot up at 240 and 320 TPS. These failed transactions are, again, caused by conflicting readsets in the MVCC. Since the average transaction latency only changes minimally when the network size is changed for the same network load, it is unlikely this is the cause of the increased error rate. Yet, the readset of a submitted transaction may not match the ledger's versioned state during commit time. This could happen when a new block was sent to 1 or 2 peers, but the remaining 3 peers endorse a transaction, without having received the new block yet. If we go back to Fabric's endorsement process in Figure 4.3, we see that the endorsement process locks the entire ledger. Thus, if nodes try to continuously endorse and commit blocks, we have a race condition in which the two processes compete. If endorsement comes before a new block is committed, the endorsements are generated over outdated values.

In Besu, the trend is the opposite: for a fixed network load, adding more nodes to the network actually reduces the error rate. Recall that errors in Besu are caused by an overflowing transaction pool. By default, this pool is set to a maximum of

4096 transactions. Although a fair share of the transactions in a pool is likely to overlap with other nodes since transactions are distributed peer-to-peer, different peers may have different transactions in their backlog, resulting in an effectively larger transaction pool. A larger transaction pool allows the network to store more transactions before they are discarded, reducing the error rate.

7.4 Findings in context of the literature

In Section 2.4, our related work already discussed the literature on previous Fabric benchmarks, especially as no papers on Besu's benchmarks are published yet. The goal of many of these papers is to push the frameworks to their absolute limits, even unrealistic for some production use cases, whereas this chapter's aim is to give the reader an intuition of how the two frameworks compare and behave for different settings on consumer hardware. There are multiple reasons why the absolute numbers of our benchmarks presented in this paper are not directly comparable with the literature. First of all, some papers use server clusters of 16 cores and/or over 64GB of RAM per peer [3, 91, 4, 35, 69, 29, 90], which is outside this thesis' budget. Secondly, all papers written at the time of writing are performed on an older version of Fabric, whereas we used Fabric v2.2. Yet, we can briefly go over the literature to compare trends, i.e. the curve of performance metrics as variables are changed.

Nasir et al. [69] investigated how a transaction's execution time, latency and throughput change as the total number of transactions increase, i.e. for 10, 100, 1.000 and 10.000 transactions. Since our benchmarks are designed to be measured over a constant number of transactions (i.e. 50.000) and a warm-up period after which the network is fully initialized, these results cannot be compared. Later, they investigate how Fabric v0.6 and v1.0's latency and throughput change as the network size changes between 1 to 20 peers. They found no significant difference, whereas we found an increased latency when the network size is larger. As the framework has been through many changes over 3 years, including a completely different consensus protocol, a direct reason for this discrepancy cannot be provided.

Similarly, many results by Thakkar et al. on Fabric v1.0 [91] are not comparable with our results as well, as they mostly look at the difference between various state databases (LevelDB and CouchDB), introducing cache mechanics, number of CPUs, and number of channels. However, they also compared the influence of different block sizes, and found that the block size has no impact on the throughput, but a higher block size does result in a higher latency, which is consistent with our results in Figure 7.9. Androulaki et al. observed the same results on Fabric v1.1.0 [3]. Another paper by Thakkar et al. [90] benchmark the Fabric v1.4 framework for their maximum throughput when the amount of organizations vary. It finds that increasing the number of organizations result in a higher throughput, most likely due to the endorsement requests being distributed over more nodes. We could not establish such connection, as we could not generate transactions faster than the network could handle. Other benchmarks are unrelated to our tests, such as different types of endorsement policies, and various proposed optimizations related to peer functionality, pipelining and others, some of which are not yet committed to the Fabric codebase at the time of writing.

Gorenflo et al. [35] benchmark Fabric v1.2 and compare it with 3 proposed optimizations. For the vanilla version of the Fabric source code, they only compare the impact of a transaction's payload size on the throughput, for which we consistently use the exact same payload size (all our benchmarks involve the same transaction type and structure). For all other comparisons, they only compare the vanilla version of Fabric with the three optimizations. Thus, they only present the outcome of a benchmark as one or two absolute numbers, rather than an overview of different configurations and environments.

Dreyer et al. [29] look at the impact of including varying amount of peers within the same organization, whereas we kept this at a fixed number of 1 peer per organization. Dreyer found that adding more peers to the same organization hurts performance by resulting in larger latency. Moreover, Dreyer et al. found that the average commit latency increases as blocks get larger, consistent with our results. Another finding of them we did not focus on, is that adding more orderers to the system has no significant impact on the system. Expanding our benchmarking towards verifying these findings is left as future work.

Chapter 8

Discussion

This discussion is structured in the following way. First, the sub-research questions are answered by summarizing some of the most interesting findings of Chapter 3 to 7. Then, we discuss the limitations of this work. Thereafter, lessons learned describes the directions a future researcher might take to add an additional blockchain framework to the comparison, or to conduct a similar study. Lastly, open issues describe future directions that both frameworks might take based on the results of this study, including limitations of Fabric and Besu and ideas for improvements.

8.1 Summarizing the Findings

8.1.1 Architecture

Chapter 3 provided an elaborate analysis of Fabric's and Besu's architecture to answer the question: "***How do the overall architectures of Fabric and Besu differ?***" Our study showed that a blockchain architecture involves many crucial moving parts, each playing an important role. Organizations using Fabric are in complete control over their internal identities through certificates. Since organizations do not exist in Besu, individual nodes and accounts need to be explicitly granted access to the network (when permissioning is enabled). If consortia decide to use Besu, a mapping between organizations and network addresses needs to be established external to the framework, whereas Fabric identifies an organization by a single (string) identifier on-chain.

Networks created using Fabric are heterogeneous by nature, both in terms of nodes as well as roles. Although the nodes of organizations are decentralized, they cannot function without an operational, possibly centralized, ordering service. This requires a clear distinction between network administrators and participants. Besu is designed as a homogeneous network in which all participants are considered equal without infrastructure-like nodes or administrative permissions.

A sharp contrast can be seen in terms of configurability as well. Fabric exposes an abundance of variables to tweak, both for the network and for nodes. This allows one to tweak the Fabric framework to a great extent and tailor it to the unique requirements of a use cases, although it requires high expertise to do so. Besu's configuration is much more limited by focusing mostly on the features to enable or disable, and takes the assumption that the default values are sufficient for most use cases (such as request timeouts, number of retries, frequency to ping other nodes). Nevertheless, it does allow nodes to tweak the more significant variables (such as Gas limits, Gas fees, transaction pool size and the maximum peers to connect with). Due to the network homogeneity in Besu, network-wide updates (such as to introduce participants or update the protocol version) must be performed unanimously by each member. In Fabric, a channel defines what organizations need to approve such changes, e.g. all participants or only a subset of administrators. Both frameworks allow for protocol updates by enabling the new version's features at some point, although Fabric enables the new features *after* all nodes have updated their client, whereas Besu announces the block at which to update *upfront*. Both frameworks provide strong and flexible access management mechanisms for the network as a whole. In addition, Besu allows individual nodes to define a whitelist of other nodes to connect to and what accounts (identities) to only accept transactions from, which is not present in Fabric.

Fabric and Besu both store blockchain state in a similar way in a key-value storage, but using different storage engines. Fabric allows the user to choose between LevelDB or CouchDB, while Besu integrates with RocksDB out-of-the-box. Both frameworks, however, offer a unique feature that the other framework does not support. Besu offers a plug-in to encrypt the ledger at rest in RocksDB. Of course, file system encryption could also be used for either framework, although it is likely that this has a significant impact on performance. By using CouchDB, Fabric uniquely allows for a replicated database set-up with rich JSON queries on the data, which is not possible using Besu's RocksDB integration.

Besu and Fabric's smart contract implementations are fundamentally different as well. Besu's smart contracts are written on a high level using Solidity or low level using Yul(+), while Fabric's smart contracts are written in general-purpose programming languages using lower-level operations. Unless restricted, any participant can deploy a smart contract in Besu, which results in the same functionality being available for all members of the network. In Fabric, a smart contract needs to be packaged, distributed to all participants off-chain, and needs to be manually installed on each node. Yet, a deployed contract's functionality may differ per node in Fabric, but not in Besu. This also means a smart contract is only able to execute another smart contract in Fabric, when these contracts have been explicitly installed locally. Since the code of a smart contract is recorded on each node's ledger in Besu, all deployed contracts have consistent code and can be interacted with without manually installing them locally on each node. While Fabric defines channel-wide policies to determine who may create, read and execute smart contracts, Besu allows the creation of a network-wide smart contract for programmatic access control. Upgrading smart contracts in Fabric requires manual approval of the policy-specified participants but is generally easy, whereas upgrading a smart contract in Besu is nontrivial (using call delegations).

8.1.2 Transaction Flow

Chapter 4 aimed to answer the research question: "***What is the transaction flow of Besu and Fabric from commit to validation?***" We observed that the transaction flow of Besu and Fabric mostly perform the same steps, but in a different order. In Besu, *requests* to invoke smart contract functionality is ordered into blocks before being executed on *all* individual peers. In Fabric, the requests to invoke smart contract functionality are executed by *some* peers before their *results* are ordered into blocks and distributed. We observed this resulted in a lot of responsibility of the actor submitting a transaction in Fabric (as confirmed in our benchmark in Section 7.3.1). In addition, Besu uses a locally-established priority ordering of transactions, whereas a transaction in Fabric is only ordered once there is agreement among OSNs that follow a first-come first-serve principle.

Some notable differences could be observed for *actor* connectivity as well. Because a Fabric actor has a lot of responsibility in terms of creating transactions, a Fabric actor requires a connection to *each* endorser. As highlighted in the chapter, this might make it infeasible for light-weight devices with limited connectivity (such as IoT) to submit transactions. Since an actor in Besu can submit a transaction to any single node, thin Besu clients could submit transactions offline to a peer, which distributes the transaction to the rest of the network. In terms of *block* delivery, both frameworks follow a similar gossip approach in which the block creator distributes the blocks to a subset of other nodes, who also distribute the blocks further.

We observed that a Besu node's ledger is always completely generated by that node's own execution of transactions, i.e. no state is received through other participants. Therefore, an honest Besu node's state is always the result of proper transaction execution according to the rules of the protocol, e.g. an adversarial participant cannot trick the node into increasing the adversary's balance or by granting it ownership of an asset. No matter if all other nodes in the network have a different state, an honest node would discard all such blocks (of which its state hash does not match). Such a guarantee on ledger integrity does not exist in Fabric, as a node does not execute the transaction, but merely applies the results generated by (possibly other) endorsers, highlighting the importance of a strong endorsement policy: Data on the Fabric network is only as trustworthy as the strength of the endorsement policy.

In terms of pure functionality, each framework has two unique features related to the transaction flow that the other does not have. Firstly, transactions in Besu may be cancelled while they are still pending by sending another transaction with a higher fee, although cancellation is on a best-effort basis. Secondly, an actor may request transaction simulation or ledger state at an earlier point in time, i.e. at a block prior to the head(s) of the blockchain. Then, Besu would keep rolling back blocks from the head until the requested block is reached. Since Besu caches every key's old and new value for each block, no transactions have to be recomputed again. Since Fabric only stores the new writeset (and not the old value), such functionality is unsupported by Fabric and requires one to recompute the ledger state from the start (and stop at the right moment!) to simulate a transaction or fetch ledger state at a previous block. The first of Fabric's unique feature is that smart contracts may be *non-deterministic*, and thus, allow for some degree of randomness in its operations, as long as the endorsements are consistent. Since Besu is *deterministic*, the available operations are restricted (through DSLs). Secondly and as a direct consequence, smart contracts in Fabric are free to interact with entities external to the network, such as databases or APIs. In contrast, a Besu smart contract only write to and read information from the ledger itself with no external interactions.

8.1.3 Consensus Protocols

In Chapter 5, we analyzed the consensus protocols of Fabric and Besu and found Fabric only recommends using Raft. Although Besu could be used with Ethash (PoW), Clique, IBFT 1.0 and IBFT 2.0, one should refrain from using PoW in a network without a cryptocurrency and IBFT 1.0 is inherently flawed in the sense that the network might become non-functional. Thus, Clique and IBFT 2.0 remain as the most viable options for permissioned networks.

Fabric's Raft is based on a leader-follower approach in which the leader individually orders blocks and continuously copies its own ordering to all followers. As such, Raft's operations on blocks are append-only, without verifying if blocks and their transactions are actually correct. Raft is not very elastic in terms of nodes: adding or removing a node needs to happen one-by-one, requiring (close to) all other nodes to be online and acknowledge the new membership before another node may be added. This addition process is prone to human error, as adding a wrong certificate of a new member might make it impossible to reach a majority on the new set of members, and thus, make the channel non-functional. Important to note is that Raft is not BFT, meaning a single Byzantine node can either have a monopoly on deciding what transactions and blocks are accepted, or make the channel non-functional using various methods. This requires a level of trust on every single entity with an OSN. Yet, operations will continue as usual when at most a minority of nodes have crashed. This consensus protocol implementation has two very strong points: (1) blocks are created as quickly as required by the network node, while not creating any blocks when there is no load and (2) all blocks returned from the ordering service are final, and as such, no (soft) forks will ever be created.

Besu's Clique consensus protocol is inherently simplistic as the only communication overhead between nodes regards the voting process on trusted signers: ordering transactions and creating blocks requires no additional communication besides the distribution of the block itself due to a deterministic function that round-robins over the list of trusted signers. By ensuring that a trusted signer may only create another block after half of the other trusted signers have created one, there is by definition consensus on the block that is *half the amount of trusted signers* before the head. This is crucial in ensuring Clique's BFT. Yet, when the trusted signer is offline during its block creation round, other trusted signers will create the block instead. This could result in a soft fork in the blockchain, meaning there could be at least 2 different heads and states of the ledger. Thus, Clique only offers probabilistic finality in which the chance of blocks becoming final increases at time progresses, often requiring users of the network to wait for a certain amount of block confirmations. This aspect of no guaranteed and immediate block finality may or may not be suitable for certain enterprise use cases. For Clique, a minimum network size of 1 node is required.

Besu's IBFT 2.0 builds upon Clique to re-use the concept of trusted validators while guaranteeing immediate block finality through a three-phase commit. In practice, it means that a two-thirds majority is required at least two times for every newly created block, showing the additional communication overhead compared to Clique. As the consensus protocol is based on PBFT [20] while being mathematically proven [82], the algorithm is BFT by tolerating up to one-third of Byzantine nodes. A significant drawback of both Besu's Clique and IBFT 2.0 is that blocks are always created at a rate predefined by the block time, meaning that if a block is full and there is a backlog, the network still sleeps and waits for the timer to create the next block. Alternatively, even when the backlog is empty and there are no transactions to order, the network creates empty blocks. A minimum network size of 3 nodes is required.

8.1.4 Data Privacy

In Chapter 6, we saw that Fabric and Besu's data privacy are very similar on a conceptual level. In both frameworks, the private data is first stored on the nodes in the PDC or private group. After this has succeeded, a public transaction is created with a hash of the private data as payload, which is ordered into blocks and distributed throughout the network. Then, once nodes receive a block that includes this private data hash, the private data is fetched and committed to the ledger. Although this concept of data privacy is very similar, we have seen that the actual implementations are very different in multiple ways. Fabric's private data implementation is completely embedded in peer nodes, whereas Besu's implementation is delegated to external Orion nodes.

The first large difference lies in the definition of private data. In Fabric, this is the readset with a list of versioned dependencies and a writeset containing the new state values for a list of keys. On the other hand, Besu defines private data as a private transaction, i.e. *request* to execute a private function. As such, a transaction in Fabric may involve both public and private operations, whereas one cannot perform public and private operations in the same message call in Besu. At the same time, a Besu's smart contract could be public or private, depending on the manner it is deployed. Because the hash recorded on the public ledger in Fabric is a hash of the actual state, transactions between multiple PDCs are possible by verifying if a claimed private state matches the value on the public ledger. In Besu, transactions between private groups are unsupported as the public hash cannot be requested from inside a smart contract, but also because the hash represents a message call, rather than private state.

In terms of integrity, we observed that Fabric applies the same validation subsystem in terms of the required endorsements. As is the case in public transactions, private data is only as secure as the endorsement policy, and thus, extra care is required to ensure the endorsement policies are either secure, or by only storing insignificant data in the private groups. As Besu nodes individually execute transactions, Besu does not have this issue. As for availability, we identified multiple interconnected issues with Besu. A transaction may only be submitted to a private group in Besu, if all nodes are operational. This is caused by the fact that a Besu node cannot pull missing private data, and as such, the members of a private group cannot be changed after they have been created. Consequently, if a Besu node's state becomes corrupt or inconsistent with other nodes in the private group, there is no (built-in) way to recover and the node is lost. Fabric solves this issue by periodically pushing and pulling private state from other peers in the same group. Another problem we encountered with Besu's data privacy is that the Orion nodes crash after about 400 transactions due to memory leaks caused by a C library that allocates memory but never releases it. Lastly, we identified that Besu nodes do not have access control, and as such, private transactions may be recovered by anyone able to open a connection to the Orion nodes, which is by default the case as all Orion nodes find each other through peer discovery. Although unfinished at the time of writing, it is expected that Besu's issues regarding availability, pulling data, permanent data loss and the data exposure vulnerability are solved by swapping Orion for Tessera.

8.1.5 Benchmarks

Chapter 7 performed a total of 98 benchmarks distributed over four experiments. These benchmarks span a number of different configurations, while trying to match the configuration settings of both frameworks as much as possible.

We looked at the out-of-the-box performance, as well as the impact of block time, block size and network size on the performance. For our out-of-the-box baseline throughput, we observed that Besu scales linearly up until 80 TPS in our reference experimental setup of one desktop computer, after which throughput starts to throttle, causing the transaction pools to overflow, which results in failed transactions. We saw that Fabric's throughput easily scales linearly up until 700 TPS, after which we can no longer generate transactions quickly enough. That said, Fabric's error rate starts to increase after 320 TPS, caused by conflicting readsets. Fabric's readset conflicts originate from the design choice to execute transactions before ordering, as well as our decision to iterate over only 500 assets. We saw that Besu's CPU usage is much lower than Fabric's, likely because the block creation frequency was the bottleneck. On the other hand, Fabric's memory usage is significantly lower than Besu's. It is likely this is caused by the fact that pending transactions are stored on a Besu node locally, and on an external OSN in Fabric. As for the amount of network traffic the nodes use, Besu uses significantly less bandwidth at lower network loads (under 80 TPS), but drastically increase thereafter. We identified that Web3.js plays a major role of Besu's increased outgoing traffic, as it creates a unique subscriber for every transaction it sends, which remain open for a long period of time once transactions start to build up (or even fail).

Blocks are created in Fabric once a block timer expires or a block's size is reached, of which its effects we clearly see in our results. Block time is mostly interesting for lower network loads (under 80 TPS) for which the block size is not filled. Then, lowering the block time results in a lower average transaction latency. At higher network loads, the effects of the block time fade as it is no longer the main trigger for creating blocks. In fact, the block limit determines when blocks are cut at higher network loads, since the block's capacity is reached before a timer expires. Lowering the block limit results in a lower latency, as blocks are cut and distributed more frequently. Since lowering block times and block limits generally reduce the average latency, this could be a good strategy to reduce the error rate. Our benchmarks showed that increasing the size of a Fabric network results in a slightly higher throughput and error rate. Although a direct cause is unclear, requiring more endorsements and a slightly slower distribution of blocks might play a role.

For Besu's benchmarks, the effects of block size and block time are much more significant, especially at higher network loads. Since blocks are created in Besu at a fixed frequency irrespective of the network load, there is always some point at which the maximum capacity of all blocks are reached. Reducing Besu's block time results in the creation of more blocks, and thus, the network is able to handle more capacity. Similarly, increasing Besu's block size results in more transactions per block, and thus, the network can process a higher network load. Since Besu's error rate is caused by overflowing transaction pools, increasing the network's capacity by lowering block time or increasing the block limit immediately reduces the error rate and latency. We have seen that setting a block limit of 4 times the default value, allows Besu to directly compete with Fabric in terms of throughput and latency for a network load of at least 320 TPS. If the size of a Besu network grows, we saw the error rate drop significantly, presumably caused by a larger collective transaction pool.

8.2 Limitations of this work

The work of this thesis is limited in a couple of different ways. First, this study focused mostly on comparing the differences of both frameworks, rather than their implications in production use cases. This means that the scope of our results are by definition of a more technical nature. Although this work sometimes hinted towards the feasibility of certain aspects of a framework, they are merely presented as considerations rather than concrete claims. More importantly, one cannot simply advice one framework over the other without knowing the concrete requirements of a specific use case. Just to name an example, one might wish to submit transactions from IoT devices for transactions that interact between multiple private groups. Although Fabric is the only framework that supports smart contract interactions between multiple private groups, not only does the actor requires internet connectivity to all other organizations specified by the endorsement policy, the action of submitting a transaction is much more computationally expensive on Fabric than on Besu. Thus, even in such a simple example, no perfect solution exists: does one choose for private data interaction, or for lightweight transaction submission? One should compose a list of requirements for a specific use case before carefully evaluating which framework to use. Instead, future work might include conducting a survey to research which technical requirements are more important for practitioners, although this does not eliminate the fact that each use case is still unique.

Secondly, although our experiments consisted of 98 unique benchmarks, the benchmarks could have been more elaborate in a variety of manners. All benchmarks were performed on a local machine with consumer-grade hardware with as primary goal to obtain an intuition of how both frameworks perform in the same setting. Especially for Fabric in which transaction generation was the bottleneck, one might want to take this further by benchmarking both frameworks on dedicated server clusters with more powerful hardware. Although this has been done in the past, these were conducted on older studies of Fabric (before version 2.2) at the time of writing. For this thesis, however, conducting these more elaborate benchmarks to push the frameworks to their ultimate limit would have exceeded the available time and budget. Our experiments looked at the impact on the performance by multiple variables present in both frameworks, but we could have looked at variables unique to each framework as well. Moreover, we were only interested in values that were directly comparable between the two frameworks. In reality, some production use cases use a subsecond block time in Fabric which we did not benchmark, as Besu does not have this option. In addition, we also did not benchmark data privacy for both frameworks, as Besu's Orion nodes suffered from memory leaks. Perhaps this could be conducted in the future once the issues are resolved or Besu's integration with Tessera is completed. Lastly, the comparison was constrained to analysing Besu and Fabric. Future studies might want to expand by looking into more frameworks.

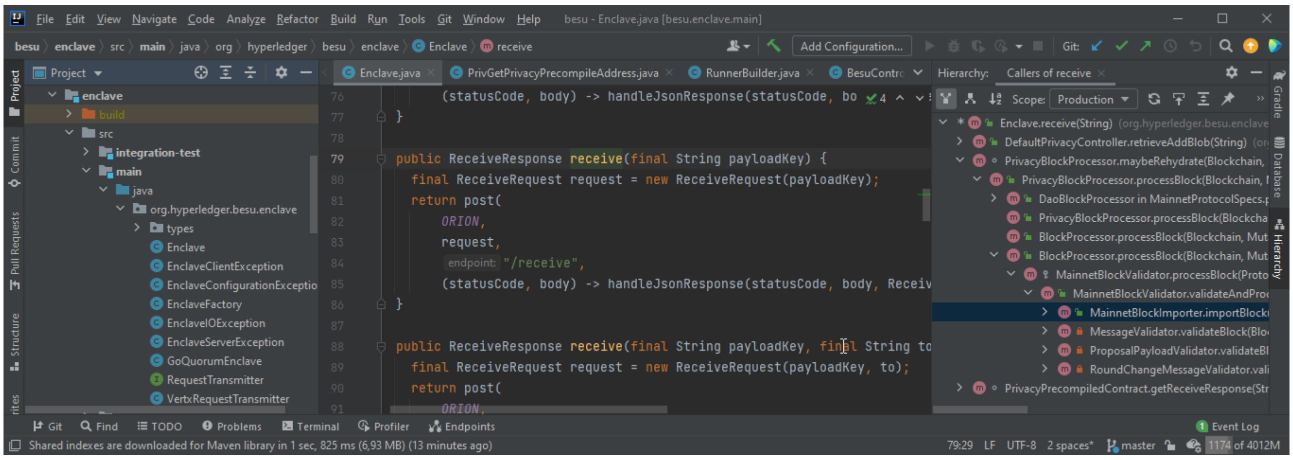
8.3 Lessons Learned

Although the task at hand, comparing two frameworks, might seem simple, the truth is far from it. Performing the work of this thesis was no easy task, and some lessons could be learnt from it. My greatest struggle regards scoping down the project. These blockchain frameworks are of such a substantial size, that it becomes very non-trivial to grasp the entire system. The classical divide-and-conquer strategy helped me to overcome this barrier by looking at the frameworks from various perspectives, while trying to abstract away from the other perspectives. I started with a top-down view on the frameworks by looking at their architecture, their basic functionalities, core principles, components and component interactions. Then, by zooming in on the more detailed aspects, such as the transaction flow and consensus protocols, the previous perspective no longer requires an explanation and one can focus on a single, scoped-down aspect. By abstracting away from data privacy in the architecture chapter, only the changes to the information already presented had to be discussed, without repeating the same information. Another challenge lies in determining the source of truth to fetch the information from. Some aspects are documented very well, such as the consensus protocols. For many other aspects, like the transaction flow, no concise and detailed documentation may exist and one has to compose its own to truly understand *how* the frameworks work, rather than simply *what* they do. For Besu, no papers have been published on documenting its architecture, transaction flow or data privacy at all. For Fabric, we have seen that some papers have been published, although all papers we could find were written for significantly older versions of the framework (prior to version 2.0), and as such, may not provide a correct representation of the truth. To overcome this, I resorted to using the source code of both frameworks as my source of truth. Using a feature-rich Integrated Development Environment (IDE) such as IntelliJ Idea[1] for Besu (Figure 8.1) and GoLand[2] for Fabric helped me create call graphs, quickly find out object hierarchies, follow references, et cetera. Although these tools helped me a lot, figuring out the internal techniques of the frameworks is still very much a complex puzzle.

8.4 Open Issues

Although the purpose of this study was not to improve either framework, some ideas have been formed on how Fabric, Besu, Web3.js and Caliper could be improved. Some of these ideas stem from issues encountered during this thesis. Although no

[1]https://www.jetbrains.com/go/
[2]https://www.jetbrains.com/idea/



```
        (statusCode, body) -> handleJsonResponse(statusCode, bo
  }

  public ReceiveResponse receive(final String payloadKey) {
    final ReceiveRequest request = new ReceiveRequest(payloadKey);
    return post(
        ORION,
        request,
        endpoint: "/receive",
        (statusCode, body) -> handleJsonResponse(statusCode, body, Receiv
  }

  public ReceiveResponse receive(final String payloadKey, final String to
    final ReceiveRequest request = new ReceiveRequest(payloadKey, to);
    return post(
        ORION,
```

```
Hierarchy: Callers of receive    Scope: Production
Enclave.receive(String) (org.hyperledger.besu.enclave
  DefaultPrivacyController.retrieveAddBlob(String) (or
  PrivacyBlockProcessor.maybeRehydrate(Blockchain,
    PrivacyBlockProcessor.processBlock(Blockchain, I
      DaoBlockProcessor in MainnetProtocolSpecs.p
      PrivacyBlockProcessor.processBlock(Blockcha
      BlockProcessor.processBlock(Blockchain, Mut
      BlockProcessor.processBlock(Blockchain, Mut
        MainnetBlockValidator.processBlock(Proto
          MainnetBlockValidator.validateAndProc
            MainnetBlockImporter.importBlock(
            MessageValidator.validateBlock(Blo
            ProposalPayloadValidator.validateBl
            RoundChangeMessageValidator.vali
  PrivacyPrecompiledContract.getReceiveResponse(Str
```

```
Shared indexes are downloaded for Maven library in 1 sec, 825 ms (6,93 MB) (13 minutes ago)
79:29  LF  UTF-8  2 spaces*  master  1174 of 4012M
```

Fig. 8.1 IntelliJ IDE

framework is (nor should be) the same, other ideas question whether a feature in one framework could possibly improve the other. Appendix B lists the issues opened with the developers of the frameworks and components.

8.4.1 Besu

Starting with Besu, we saw many issues regarding data privacy in Orion in Chapter 6. Before data privacy could be seriously considered for production networks, either (1) Orion's memory leak needs to be resolved, access control should be implemented to only allow the fetching of private data by members in the same private group, and data pulling between Orion nodes should be supported to reduce its high availability requirement and to prevent a single node from having an inconsistent unrecoverable state. Or (2), alternatively, Besu's migration to Tessera, which does support the aforementioned features, would have to be completed. For now, however, one would likely resort to creating multiple distinct private *networks*, rather than private *groups*. As was discussed in Section 3.3.6, however, this is still an open issue as well for which research is still ongoing. A further improvement could be to research how Besu's private transactions could be made compatible with the Ethereum MainNet, of which first steps could be to implement (or enable) block rollbacks for private data as discussed in Section 6.2.4.

Another auxiliary component of Besu, EthSigner, could also be improved. A single EthSigner instance may only define the endpoint of one Besu node. Ideally, if the specified Besu node is offline, the EthSigner instance should forward requests to a different node. To achieve high availability using the current components in combination with EthSigner, one would require multiple EthSigner instances per node. Future improvements could allow EthSigner instances to define multiple Besu nodes, such that if one is offline, it connects to another.

Although Besu nodes themselves have access control, in contrary to Orion nodes, access control of Besu nodes is very simplistic. This could be a strong point of Besu as it allows one to quickly configure the network and get up and running, but enterprises might require more sophisticated techniques. For instance, the JWT allows users to authenticate themselves at a node and receive an access token in return. Perhaps, this could be expanded to implement the OAuth protocol [43], allowing users to request authorization grants and access tokens on standalone enterprise servers, before communicating with nodes. This could make it possible for users to request a single token, which they can use with all nodes of the user's organization.

We also learnt from Fabric that a single block time and block limit setting can be very flexible for varying network loads, whereas Besu's block time and limit have to be tweaked for the expected load. By implementing a similar technique in which a block is created as soon as its capacity is reached or a timer expires, while immediately proceeding with the next block, Besu's current bottleneck for our benchmarks would be resolved. That said, Besu's Clique and IBFT consensus protocols would likely require significant changes as they rely on predictable block times to decide if another trusted signer should take over when the expected signer does not produce a block.

Another idea was inspired by how Besu changes functionality of the Ethereum protocol by enabling them at a certain (milestone) block. These often involve breaking changes, including the entire structure of transactions and blocks. A primary difficulty in switching consensus protocols in Besu, is caused by the fact that consensus protocol-specific headers are attached to each block, making a different consensus protocol unable to parse the old blocks. By developing a mechanism similarly to how breaking EIPs are enabled, perhaps one would be able to easily switch consensus protocols at some block in the future. For instance, one could define a mapping that at block `X`, consensus protocol `Y` would be enabled. Again, this is just a thought, but could be worth looking into.

Fabric's transaction flow has, without doubt, a significantly positive impact on its fast throughput as no transactions need to be executed by peers receiving a block. Naturally, the next question to ask would be whether such a transaction flow could work for Besu (or even the Ethereum MainNet), in which transactions are executed before blocks are created. For permissioned networks, this should in theory not be much of a problem: if two-thirds of all participants agree on the same execution results, the remaining one-third could apply those changes without too much of a risk. Although executing all transactions by *each* node is obviously more secure than by letting *two-thirds* of the network decide the state of the ledger, trading off security for performance might be worth it. Since nodes are explicitly admitted in a permissioned network, one could not simply spin up a lot of nodes to "become" the two-third majority without network agreement. For public Ethereum networks, however, this *is* the case: one person could spin up a lot of different nodes to try to write illegal changes. Instead of requiring two-thirds of the network *participants* to endorse, one could perhaps introduce staking in which participants with a total of two-thirds of the *staked value* must have endorsed the transaction[3]. Such transaction flow would, as we have seen in Fabric, technically allow for nondeterministic transactions. In addition, an *endorsing fee* would incentivize participants to stake Ether and become an endorser. Again, these are mere thoughts that occurred to me for future research, and are not proven or guaranteed to work out as expected in any way.

8.4.2 Fabric

Despite ongoing projects such as Hyperledger Cactus and Quilt, as discussed in Section 3.3.6, smart contract interactions between multiple Fabric networks is very much an ongoing field of research, just like with Besu.

We briefly mentioned that Besu has a plugin to encrypt all ledger data, and saw Fabric users would have to resort to file system encryption. For various reasons, one might not want to encrypt an entire drive or partition. Thus, evaluating whether encryption on the software-level could be beneficial to the Fabric framework might be worth investigating. We also saw that Besu supports operations to simulate a transaction or fetch ledger state at a block before the current head. Perhaps, some use cases might desire insights in the history of the (entire) ledger's state. Two methods for this exist in Besu, both of which could potentially work for Fabric: (1) to enable *archiving* on a node such that it caches the entire ledger's state after each block, or (2) by expanding the transaction's contents by including the old value of each key, e.g. (key, readVersion, *oldValue*, newValue)-tuples[4].

Another strong feature of Besu is that the process of submitting a transaction is much simpler, involving less steps and connections, whereas Fabric's actor has to perform a lot of tasks. To make an actor's client thinner, much of the submitting logic could possibly be delegated to either a peer or some form of *client-proxy*, such that an actor could submit a transaction in more of a fire-and-forget manner.

In Chapter 6, we discussed that one should use a PDC carefully: requiring too few endorsements results in reduced trustworthiness of the data, whereas too many might require too many participants and eliminate the privacy purpose

[3] Note that this is a completely different transaction flow, and thus, is different from the upcoming Proof of Stake EIP

[4] Although Besu does not include a writeset in the *transaction* itself, recall from Section 4.3.4 that it *does* cache every transaction's old- and new value locally on each node, now used for rollbacks and roll-forwards

of PDCs. Although we mentioned some general solutions, e.g. not storing critical data in PDCs, require a notary's endorsement, or simply include a large amount of organizations, none of these solutions are perfect. In the same section, we described how it might be worth investigating read-only PDCs in which values can only transferred between PDCs but not changed, in combination with regular (producer/manufacturer) PDCs in which values can be created. Although this only solves the security aspect for create-and-transfer smart contracts, it might be a stepping stone for further improvements.

8.4.3 Other Components

For Caliper, we already proposed and committed two improvements to (1) fix the Docker resource monitor and (2) a new feature to send transactions from multiple Ethereum accounts in the same benchmarks. Yet, one of our changes was not committed (due to hardcoding) for disabling the waiting time in between rounds as to enable a warm-up/initialization round. Implementing this as a toggle between rounds could perhaps be useful for other use cases as well. Another feature that Caliper lacked, was that it only exports aggregated benchmark results, i.e. minimum, maximum, and average latency. For more complicated studies, it could be useful if Caliper exported more fine-grained results, such as the throughput/resource consumption for a configurable interval (such as every second), or the exact submission and commit times per transaction. In our experiments (Section 7.3.1), we observed that Web3.js creates a unique subscriber for every single transaction being sent. Ideally, this client library could be improved to only open at most 1 connection such that the same block headers are not submitted numerous times to the same client, e.g. by making the subscriber a singleton.

Chapter 9

Conclusion

The primary goal of this work was to answer the following research question: "***What are the differences between Hyperledger Fabric and Hyperledger Besu, as well as their advantages and disadvantages?***" and answered this question by dividing the problem in five different sub-research questions, each focusing on one aspect or perspective of the framework.

In our architecture chapter, we looked at the building blocks of each blockchain framework and found Besu is fundamentally simpler in terms of operations, configurability, network composition and smart contract design. At the same time, Fabric has more powerful features for organizations such as identity management, extensive configurations and policies, allowing one to tailor the blockchain to individual use cases. Our chapter on transaction flow visualized the entire process of submitting and committing a transaction from start to end, revealing that actors in Fabric have to perform relatively many tasks, the ledger could be manipulated when the endorsement policy is insecure, but allows for undeterministic smart contracts. Although Besu's transaction flow is more intuitive and less prone to manipulation, transactions have to be executed more frequently. Besu uniquely allows for the cancellation of pending transactions and state operations on blocks older than the current head. A chapter on consensus protocols investigated how agreement on the state of the ledger is reached in both frameworks, in which we saw that Besu's consensus protocols are Byzantine Fault Tolerant but creates blocks at a fixed rate regardless of the network load, whereas Fabric is only Crash Fault Tolerant but only creates as many blocks as required to handle the current load. Our chapter on data privacy showed how multiple private groups are able to interact with each other in Fabric but requires great care. Private groups in Besu are isolated namespaces that regard the public ledger as read-only. Although Besu designed data privacy similarly as Fabric on a conceptual level of the transaction flow, its implementation suffers from numerous flaws that make it undesirable for production use cases. Lastly, our benchmarks revealed that Fabric performs much better out-of-the-box in terms of throughput and block latency, uses significantly less memory, but is heavier on the CPU. When transactions start to fail, Besu's outgoing traffic starts to skyrocket due to inefficiencies with Web3.js that is relied on. Both frameworks can be tweaked for a stable throughput of at least 320 TPS and a transaction latency below 1.5 seconds on a consumer-grade desktop computer.

Although this work has presented each framework in great detail and compared both frameworks on numerous aspects, neither framework is evidently the best choice for *all* use cases. It is therefore important that one decides on a list of weighted requirements before choosing between either framework. In addition, the information presented in this paper may not just be valuable for architects looking for a framework for their use case, but also for researchers interested in learning more about either framework, strategists questioning which framework is worth investing resources on for their firm's future endeavors, and blockchain developers hoping to gain competitive insights in learning how other projects solve certain problems. And although one chapter could be more interesting than others for certain types of readers, we believe that the entirety of this thesis provides a sufficiently complete and in-depth comparison of both frameworks to bring value to a wide variety of stakeholders.

References

[1] Ailijiang, A., Charapko, A., and Demirbas, M. (2016). Consensus in the cloud: Paxos systems demystified. In *2016 25th International Conference on Computer Communication and Networks (ICCCN)*, pages 1–10. IEEE.

[2] Allison, I. (2020). Consensys acquires jpmorgan's quorum blockchain. https://www.coindesk.com/consensys-acquires-jp-morgan-quorum-blockchain.

[3] Androulaki, E., Barger, A., Bortnikov, V., Cachin, C., Christidis, K., De Caro, A., Enyeart, D., Ferris, C., Laventman, G., Manevich, Y., et al. (2018). Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the thirteenth EuroSys conference*, pages 1–15.

[4] Androulaki, E., De Caro, A., Neugschwandtner, M., and Sorniotti, A. (2019). Endorsement in hyperledger fabric. In *2019 IEEE International Conference on Blockchain (Blockchain)*, pages 510–519. IEEE.

[5] Antar, A. (2020). Scaling smart contracts on hedera hashgraph. https://medium.com/hearo-fm/scaling-smart-contracts-on-hedera-hashgraph-47a3c90de218.

[6] Back, A., Corallo, M., Dashjr, L., Friedenbach, M., Maxwell, G., Miller, A., Poelstra, A., Timón, J., and Wuille, P. (2014). Enabling blockchain innovations with pegged sidechains. https://blockstream.com/sidechains.pdf.

[7] Baird, L. (2016). The swirlds hashgraph consensus algorithm: Fair, fast, byzantine fault tolerance. *Swirlds Tech Reports SWIRLDS-TR-2016-01, Tech. Rep.*

[8] Baird, L., Gross, B., and Donald, T. (2020). Hedera consensus service. *Hedera Hashgraph*.

[9] Baird, L., Harmon, M., and Madsen, P. (2018). Hedera: A governing council & public hashgraph network. *The trust layer of the internet, whitepaper*, 1:1–97.

[10] Balagurusamy, V. S., Cabral, C., Coomaraswamy, S., Delamarche, E., Dillenberger, D. N., Dittmann, G., Friedman, D., Gökçe, O., Hinds, N., Jelitto, J., et al. (2019). Crypto anchors. *IBM Journal of Research and Development*, 63(2/3):4–1.

[11] Batubara, F. R., Ubacht, J., and Janssen, M. (2018). Challenges of blockchain technology adoption for e-government: a systematic literature review. In *Proceedings of the 19th Annual International Conference on Digital Government Research: Governance in the Data Age*, pages 1–9.

[12] Behnke, K. and Janssen, M. (2020). Boundary conditions for traceability in food supply chains using blockchain technology. *International Journal of Information Management*, 52:101969.

[13] Belchior, R., Vasconcelos, A., Guerreiro, S., and Correia, M. (2020). A survey on blockchain interoperability: Past, present, and future trends. *arXiv preprint arXiv:2005.14282*.

[14] Besu, H. (2021). Privacy. https://besu.hyperledger.org/en/stable/Concepts/Privacy/Privacy-Overview/.

[15] Bible, W., Raphael, J., Taylor, P., and Valiente, I. O. (2017). Blockchain technology and its potential impact on the audit and assurance profession. *Aicpa. org*.

[16] Bissias, G., Levine, B. N., Ozisik, A. P., and Andresen, G. (2016). An analysis of attacks on blockchain consensus. *arXiv preprint arXiv:1610.07985*.

[17] Brewer, E. A. (2000). Towards robust distributed systems. In *PODC*, volume 7, pages 343477–343502. Portland, OR.

[18] Brown, D. R. (2010). Sec 2: Recommended elliptic curve domain parameters. *Standars for Efficient Cryptography*.

[19] Buterin, V. (2016). R3 report-chain interoperability. *R3 Res*.

[20] Castro, M., Liskov, B., et al. (1999). Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186.

[21] Chandra, T. D., Griesemer, R., and Redstone, J. (2007). Paxos made live: an engineering perspective. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 398–407.

[22] Chen, J., Xia, X., Lo, D., Grundy, J., and Yang, X. (2020). Maintaining smart contracts on ethereum: Issues, techniques, and future challenges. *arXiv preprint arXiv:2007.00286*.

[23] Coglio, A. (2020). Ethereum's recursive length prefix in acl2. *arXiv preprint arXiv:2009.13769*.

[24] ConsenSys (2018). Scaling consensus for enterprise: Explaining the ibft algorithm. https://consensys.net/blog/enterprise-blockchain/scaling-consensus-for-enterprise-explaining-the-ibft-algorithm/.

[25] ConsenSys (2019). https://consensys.net/blog/news/another-day-another-consensus-algorithm-why-ibft-2-0/.

[26] Darragh, J. J., Cleary, J. G., and Witten, I. H. (1993). Bonsai: a compact representation of trees. *Software: Practice and Experience*, 23(3):277–291.

[27] De Angelis, S., Aniello, L., Baldoni, R., Lombardi, F., Margheri, A., and Sassone, V. (2018). Pbft vs proof-of-authority: Applying the cap theorem to permissioned blockchain. *Italian Conference on Cybersecurity*.

[28] Deshpande, A., Stewart, K., Lepetit, L., and Gunashekar, S. (2017). Distributed ledger technologies/blockchain: Challenges, opportunities and the prospects for standards. *Overview report The British Standards Institution (BSI)*, 40:40.

[29] Dreyer, J., Fischer, M., and Tönjes, R. (2020). Performance analysis of hyperledger fabric 2.0 blockchain platform. In *Proceedings of the Workshop on Cloud Continuum Services for Smart IoT Systems*, pages 32–38.

[30] Ekparinya, P., Gramoli, V., and Jourjon, G. (2018). Double-spending risk quantification in private, consortium and public ethereum blockchains. *arXiv preprint arXiv:1805.05004*.

[31] El Ioini, N. and Pahl, C. (2018). A review of distributed ledger technologies. In *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*, pages 277–288. Springer.

[32] Entriken, W., Shirley, D., Evans, J., and Sachs, N. (2017). Eip-721: Erc-721 non-fungible token standard. https://eips.ethereum.org/EIPS/eip-721.

[33] Faisal, E., Golash, R., and Ma, K. (2021). Ibft consensus. https://whitepaper.ledgerium.io/architecture-blockchain/ibft.

[34] Frizzo-Barker, J., Chow-White, P. A., Adams, P. R., Mentanko, J., Ha, D., and Green, S. (2020). Blockchain as a disruptive technology for business: A systematic review. *International Journal of Information Management*, 51:102029.

[35] Gorenflo, C., Lee, S., Golab, L., and Keshav, S. (2020). Fastfabric: Scaling hyperledger fabric to 20 000 transactions per second. *International Journal of Network Management*, 30(5):e2099.

[36] Greenspan, G. (2019). Sap pharma case study and interview. https://www.multichain.com/blog/2019/10/sap-pharma-case-study-interview/.

[37] Gupta, I., Birman, K. P., and Van Renesse, R. (2002). Fighting fire with fire: using randomized gossip to combat stochastic scalability limits. *Quality and Reliability Engineering International*, 18(3):165–184.

[38] Gupta, S., Hellings, J., Rahnama, S., and Sadoghi, M. (2020). Building high throughput permissioned blockchain fabrics: challenges and opportunities. *Proceedings of the VLDB Endowment*, 13(12):3441–3444.

[39] Hamida, E. B., Brousmiche, K. L., Levard, H., and Thea, E. (2017). Blockchain for enterprise: overview, opportunities and challenges. In *The Thirteenth International Conference on Wireless and Mobile Communications (ICWMC 2017)*.

[40] Handagama, S. (2020). Around 100 italian banks are officially on a blockchain. https://www.coindesk.com/italian-banking-association-100.

[41] Hankerson, D., Menezes, A. J., and Vanstone, S. (2006). *Guide to elliptic curve cryptography*. Springer Science & Business Media.

[42] Hardjono, T., Lipton, A., and Pentland, A. (2019). Toward an interoperability architecture for blockchain autonomous systems. *IEEE Transactions on Engineering Management*, 67(4):1298–1309.

[43] Hardt, D. et al. (2012). The oauth 2.0 authorization framework.

[44] Hasan, H. R., Salah, K., Jayaraman, R., Yaqoob, I., and Omar, M. (2020). Blockchain architectures for physical internet: A vision, features, requirements, and applications. *IEEE Network*.

[45] Helliar, C. V., Crawford, L., Rocca, L., Teodori, C., and Veneziani, M. (2020). Permissionless and permissioned blockchain diffusion. *International Journal of Information Management*, 54:102136.

[46] Herlihy, M. (2018). Atomic cross-chain swaps. In *Proceedings of the 2018 ACM symposium on principles of distributed computing*, pages 245–254.

[47] Higgins, S. (2016). Jp morgan is quietly developing a private ethereum blockchain. https://www.coindesk.com/jpmorgan-ethereum-blockchain-quorum.

[48] Housley, R., Ford, W., Polk, W., and Solo, D. (1999). Internet x. 509 public key infrastructure certificate and crl profile. Technical report, RFC 2459, January.

[49] ING, W. (2019). Voltron trial sees major development of blockchain trade application. https://www.ingwb.com/themes/distributed-ledger-technology-articles/voltron-trial-sees-major-development-of-blockchain-trade-application.

[50] Javaid, H., Hu, C., and Brebner, G. (2019). Optimizing validation phase of hyperledger fabric. In *2019 IEEE 27th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 269–275. IEEE.

[51] Kamath, R. (2018). Food traceability on blockchain: Walmart's pork and mango pilots with ibm. *The Journal of the British Blockchain Association*, 1(1):3712.

[52] Kan, L., Wei, Y., Muhammad, A. H., Siyuan, W., Linchao, G., and Kai, H. (2018). A multiple blockchains architecture on inter-blockchain communication. In *2018 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, pages 139–145. IEEE.

[53] Kolehmainen, T., Laatikainen, G., Kultanen, J., Kazan, E., and Abrahamsson, P. (2020). Using blockchain in digitalizing enterprise legacy systems: An experience report. In *International Conference on Software Business*, pages 70–85. Springer.

[54] Kwon, J. (2014). Tendermint: Consensus without mining. *Draft v. 0.6, fall*, 1(11).

[55] Kwon, J. and Buchman, E. (2019). Cosmos whitepaper.

[56] Köller, T. v. (2017). Raft explained – part 1/3: Introduction to the consensus problem. https://blog.container-solutions.com/raft-explained-part-1-the-consenus-problem.

[57] Lacity, M. C. (2018). Addressing key challenges to making enterprise blockchain applications a reality. *MIS Quarterly Executive*, 17(3):201–222.

[58] Lamport, L. et al. (2001). Paxos made simple. *ACM Sigact News*, 32(4):18–25.

[59] Latif-Shabgahi, G., Bass, J., and Bennett, S. (2004). A taxonomy for software voting algorithms used in safety-critical systems. *IEEE Transactions on Reliability*, 53(3):319–328.

[60] Liu, M., Wu, K., and Xu, J. J. (2019). How will blockchain technology impact auditing and accounting: Permissionless versus permissioned blockchain. *Current Issues in Auditing*, 13(2):A19–A29.

[61] Luu, L., Narayanan, V., Zheng, C., Baweja, K., Gilbert, S., and Saxena, P. (2016). A secure sharding protocol for open blockchains. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 17–30.

[62] MarketScreener (2020). Jpmorgan chase : J.p. morgan adds new features to newly branded liink^{SM}: Marketscreener. https://www.marketscreener.com/quote/stock/JPMORGAN-CHASE-CO-37468997/news/JPMorgan-Chase-J-P-Morgan-Adds-New-Features-to-Newly-Branded-Liink-8480-31626696/.

[63] McCrank, J. (2015). Nasdaq partners with chain to bring blockchain to private market. https://www.reuters.com/article/nasdaq-blockchain-idUSL1N0Z92I720150624.

[64] Montgomery, H., Borne-Pons, H., Hamilton, J., Bowman, M., Somogyvari, P., Fujimoto, S., Takeuchi, T., Kuhrt, T., and Belchior, R. (2020). Hyperledger cactus whitepaper. *External Links: Link Cited by*, 2:2–4.

[65] Morkunas, V. J., Paschen, J., and Boon, E. (2019). How blockchain technologies impact your business model. *Business Horizons*, 62(3):295–306.

[66] Morley, M. (2010). Json-rpc 2.0 specification.

[67] Moubarak, J., Filiol, E., and Chamoun, M. (2018). On blockchain security and relevant attacks. In *2018 IEEE Middle East and North Africa Communications Conference (MENACOMM)*, pages 1–6. IEEE.

[68] Nakamoto, S. (2008). Bitcoin: A peer-to-peer electronic cash system. Technical report, Manubot.

[69] Nasir, Q., Qasse, I. A., Abu Talib, M., and Nassif, A. B. (2018). Performance analysis of hyperledger fabric platforms. *Security and Communication Networks*, 2018.

[70] Nevile, C., Polzer, G., Coote, R., Noble, G., Burnett, D., and Hyland-Wood, D. (2018). Enterprise ethereum alliance client specification v6. *Enterprise Ethereum Alliance 2020. Available online: https://entethalliance.org/wp-content/uploads/2020/11/EEA_Enterprise_Ethereum_Client_Specification_v6.pdf (accessed on 12 February 2021).*

[71] Olson, K., Bowman, M., Mitchell, J., Amundson, S., Middleton, D., and Montgomery, C. (2018). Sawtooth: an introduction. *The Linux Foundation*.

[72] Ongaro, D. and Ousterhout, J. (2014). In search of an understandable consensus algorithm. In *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*, pages 305–319.

[73] Podgorelec, B., Turkanović, M., and Šestak, M. (2020). A brief review of database solutions used within blockchain platforms. In *International Congress on Blockchain and Applications*, pages 121–130. Springer.

[74] Polge, J., Robert, J., and Le Traon, Y. (2020). Permissioned blockchain frameworks in the industry: A comparison. *ICT Express*.

[75] Popov, S. (2018). The tangle. *White paper*, 1:3.

[76] Popov, S., Moog, H., Camargo, D., Capossele, A., Dimitrov, V., Gal, A., Greve, A., Kusmierz, B., Mueller, S., Penzkofer, A., et al. (2020). The coordicide. *Accessed Jan*, pages 1–30.

[77] RDW (2018). 10% duitse importauto's heeft teruggedraaide teller. https://www.rdw.nl/over-rdw/nieuws/2018/10-procent-duitse-importautos-heeft-teruggedraaide-teller.

[78] Reed, D., Law, J., and Hardman, D. (2016). The technical foundations of sovrin. *Technical report, Sovrin, 2016*.

[79] Robinson, P. (2019). The merits of using ethereum mainnet as a coordination blockchain for ethereum private sidechains. *arXiv preprint arXiv:1906.04421*.

[80] Robinson, P., Hyland-Wood, D., Saltini, R., Johnson, S., and Brainard, J. (2019). Atomic crosschain transactions for ethereum private sidechains. *arXiv preprint arXiv:1904.12079*.

[81] Saltini, R. and Hyland-Wood, D. (2019a). Correctness analysis of ibft. *arXiv preprint arXiv:1901.07160*.

[82] Saltini, R. and Hyland-Wood, D. (2019b). Ibft 2.0: A safe and live variation of the ibft blockchain consensus protocol for eventually synchronous networks. *arXiv preprint arXiv:1909.10194*.

[83] Sayeed, S. and Marco-Gisbert, H. (2019). Assessing blockchain consensus and security mechanisms against the 51% attack. *Applied Sciences*, 9(9):1788.

[84] Singh, A., Click, K., Parizi, R. M., Zhang, Q., Dehghantanha, A., and Choo, K.-K. R. (2020). Sidechain technologies in blockchain networks: An examination and state-of-the-art review. *Journal of Network and Computer Applications*, 149:102471.

[85] Soelman, M. (2019). Hyperledger fabric: A study of endorsement policies for supply chains. *Student Theses Faculty of Science and Engineering*.

[86] Soelman, M., Andrikopoulos, V., Pérez, J. A., Theodosiadis, V., Goense, K., and Rutjes, A. (2020). Hyperledger fabric: Evaluating endorsement policy strategies in supply chains. In *2020 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPS)*, pages 145–152. IEEE.

[87] Soohoo, S. (2019). New idc spending guide sees strong growth in blockchain solutions leading to $15.9 billion market in 2023. https://www.idc.com/getdoc.jsp?containerId=prUS45429719.

[88] Soohoo, S. (2020). Blockchain solutions will continue to see robust investments, led by banking and manufacturing, according to new idc spending guide. https://www.idc.com/getdoc.jsp?containerId=prUS46850820.

[89] Szilagyi, P. (2017). Eip-225: Clique proof-of-authority consensus protocol. https://eips.ethereum.org/EIPS/eip-225.

[90] Thakkar, P. and Nathan, S. (2020). Scaling hyperledger fabric using pipelined execution and sparse peers. *arXiv preprint arXiv:2003.05113*.

[91] Thakkar, P., Nathan, S., and Viswanathan, B. (2018). Performance benchmarking and optimizing hyperledger fabric blockchain platform. In *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 264–276. IEEE.

[92] Thomas, S. and Schwartz, E. (2015). A protocol for interledger payments. *URL https://interledger. org/interledger. pdf*.

[93] Tumasjan, A. and Beutel, T. (2019). Blockchain-based decentralized business models in the sharing economy: A technology adoption perspective. In *Business transformation through blockchain*, pages 77–120. Springer.

[94] Vazirani, A. A., O'Donoghue, O., Brindley, D., and Meinert, E. (2020). Blockchain vehicles for efficient medical record management. *NPJ digital medicine*, 3(1):1–5.

[95] Walport, M. et al. (2016). Distributed ledger technology: Beyond blockchain. *UK Government Office for Science*, 1:1–88.

[96] Wood, G. (2016). Polkadot: Vision for a heterogeneous multi-chain framework. *White Paper*.

[97] Wood, G. et al. (2014). Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32.

[98] Xiao, Y., Zhang, N., Lou, W., and Hou, Y. T. (2020). A survey of distributed consensus protocols for blockchain networks. *IEEE Communications Surveys & Tutorials*, 22(2):1432–1465.

Appendix A

Hyperledger Fabric

A.1 Asset Creation Chaincode Examples

```java
import org.hyperledger.fabric.contract.Context;
import org.hyperledger.fabric.contract.ContractInterface;

public final class AssetTransfer implements ContractInterface {
    private final Genson genson = new Genson();

    @Transaction(intent = Transaction.TYPE.SUBMIT)
    public Asset CreateAsset(final Context ctx, final String assetID,
        final String color) {
        ChaincodeStub stub = ctx.getStub();
        Asset asset = new Asset(assetID, color);
        String assetJSON = genson.serialize(asset);
        stub.putStringState(assetID, assetJSON);
        return asset;
    }
}
```

Listing A.1 Java Smart Contract for asset creation

```javascript
const { Contract } = require('fabric-contract-api');

class AssetTransfer extends Contract {
    async CreateAsset(ctx, id, color) {
        const asset = {
            ID: id,
            Color: color
        };
        ctx.stub.putState(id, Buffer.from(JSON.stringify(asset)));
        return JSON.stringify(asset);
    }
}
```

Listing A.2 JavaScript Smart Contract for asset creation

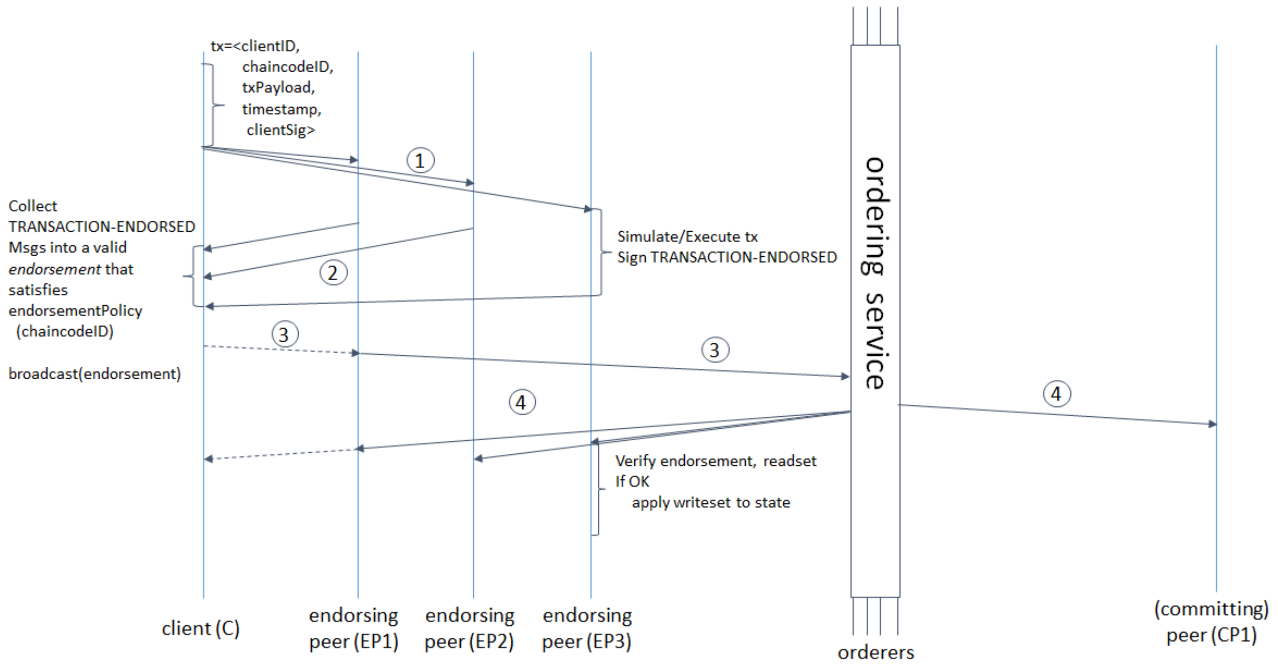
```
import (
  "encoding/json"
  "fmt"
  "github.com/hyperledger/fabric-contract-api-go/contractapi"
)

type SmartContract struct {
  contractapi.Contract
}

func(s *SmartContract) CreateAsset(ctx contractapi.TransactionContextInterface,
    id string, color string) error {
  asset := Asset{
    ID:             id,
    Color:          color
  }
  assetJSON, err := json.Marshal(asset)
  return ctx.GetStub().PutState(id, assetJSON)
}
```

Listing A.3 GoLang Smart Contract for asset creation

A.2 Transaction Flow Sequence Diagram



tx=<clientID, chaincodeID, txPayload, timestamp, clientSig>
1
Collect TRANSACTION-ENDORSED Msgs into a valid *endorsement* that satisfies endorsementPolicy (chaincodeID)
2
Simulate/Execute tx
Sign TRANSACTION-ENDORSED
3
broadcast(endorsement)
3
ordering service
4
4
Verify endorsement, readset
If OK
apply writeset to state
client (C)
endorsing peer (EP1)
endorsing peer (EP2)
endorsing peer (EP3)
orderers
(committing) peer (CP1)

Fig. A.1 Sequence Diagram of Fabric's transaction flow provided by the documentation[1].

[1]https://hyperledger-fabric.readthedocs.io/en/release-2.2/txflow.html

A.3 Public Key Infrastructure and Certificate Authorities

PKI is a well-known concept in the domain of cryptography and resulted in the X.509 standard aiming to provide "deterministic, automated identification, authentication, access control, and authorization functions" [48]. PKI relies on *digital signatures*. A private key is a cryptographic code only known to the owner of the key, whereas the public key consists of a cryptographic code that is widely known and distributed to others. The private key can be used to sign any message to produce a *signed message*. Then, if this signed message is shared with others, they are reliably able to verify whether this message originated from the person claiming to have signed the message by signing the same message using the originator's public key and comparing their signatures. The receiver of the message is thus able to find out if the message was modified or constructed by another person (without the correct private key). This flow is illustrated in Figure A.2. Consequently, if a private key is compromised by an adversary, the adversary can sign messages on the original owner's behalf. Then, the public key should no longer be trusted.

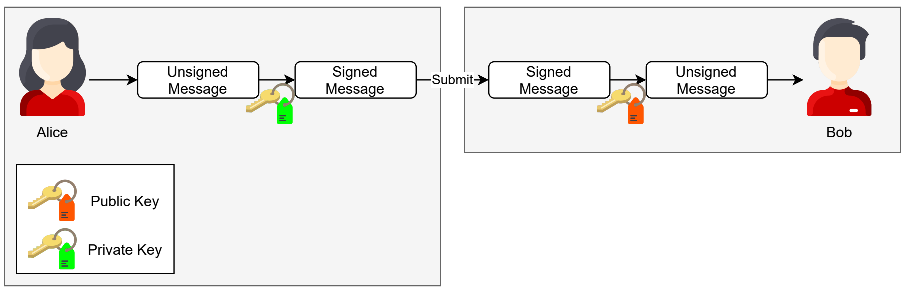


Fig. A.2 Message Signing

At the foundation of a PKI are *CAs*. All CAs have their own public and private key pair that may be used to sign the public key of another *intermediate CA* to recognize it as authentic and valid, creating a hierarchy of authorities. A root CA is an authority that has not been recognized (i.e. signed and trusted) by a higher authority. In addition to signing other intermediate authorities, CAs can issue certificates to individuals with a unique public key. These signed certificates are called *digital identities* and are used by individuals to authenticate themselves as a member of an organization. When an individual's identity has been compromised or should be revoked for another reason, the digital identity should be put on a *Certificate Revocation List*. This is essentially a blacklist of digital identities.

When a CA is compromised, the certificate of the CA itself should be put on the revocation list, which will also invalidate all credentials issued by this CA: other CAs on the same level or above are unaffected. Obviously, when the root CA is revoked, every single CAs and credentials in that hierarchy are invalidated.

A.4 Grammar, Examples and Demo of Fabric's Policies

Policies can be written using two different techniques: a *Signature Policy* or an *ImplicitMeta Policy*. The former specifies one or more exact organizations that must sign a change in combination with the operators `OR`, `AND`, or `NOutOf` and are self-describing. Moreover, operators can be nested. The resulting grammar can be written using Backus-Naur Form (BNF) in combination with Regex as shown below. The curious reader is invited to try the grammar out and experiment with it[2] and an example is illustrated in Figure A.3.

```
⟨role⟩          ::= “member”
                 |  “peer”
                 |  “admin”
```

[2]Make sure to select the "policy" non-terminal: https://bit.ly/3qLOOZb. You may use the Generate Random button to see valid examples

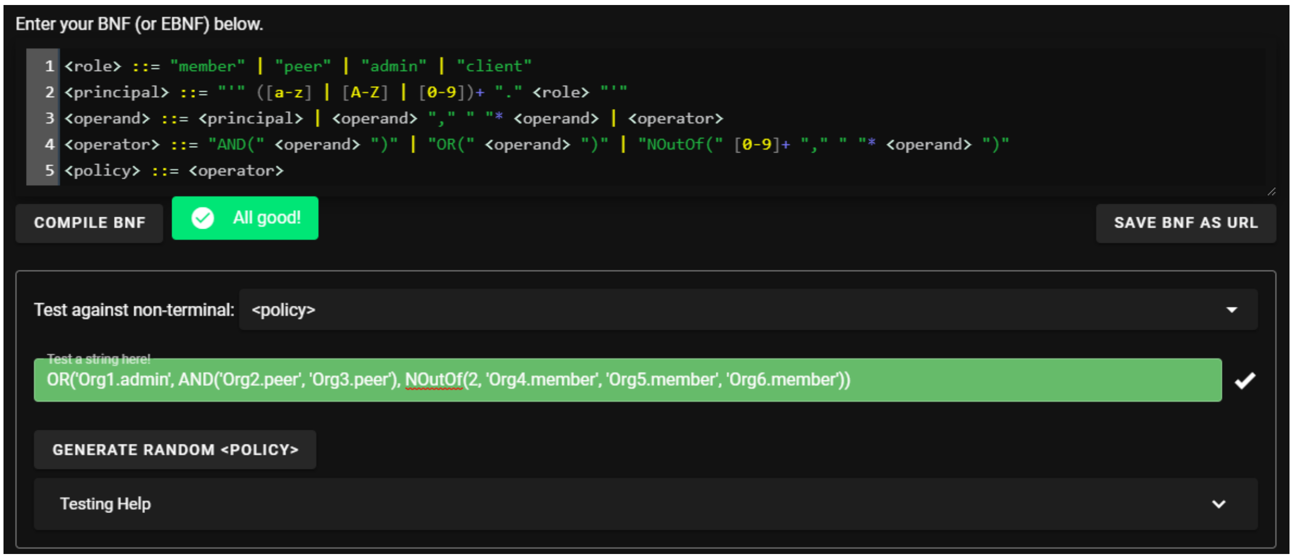
```
                    |   “client”
                    |   “orderer”

⟨principal⟩        ::= “ ’ ” ([a-z] | [A-Z] | [0-9])+ “.” ⟨role⟩ “ ’ ”

⟨operand⟩          ::= ⟨principal⟩
                    |   ⟨operand⟩ “,” “ ”* ⟨operand⟩
                    |   ⟨operator⟩

⟨operator⟩         ::= “AND(” ⟨operand⟩ “)”
                    |   “OR(” ⟨operand⟩ “)”
                    |   “NOutOf(” [0-9]+ “,” “ ”* ⟨operand⟩ “)”

⟨policy⟩           ::= ⟨operator⟩
```



Enter your BNF (or EBNF) below.

```
<role> ::= "member" | "peer" | "admin" | "client"
<principal> ::= "'" ([a-z] | [A-Z] | [0-9])+ "." <role> "'"
<operand> ::= <principal> | <operand> "," " "* <operand> | <operator>
<operator> ::= "AND(" <operand> ")" | "OR(" <operand> ")" | "NOutOf(" [0-9]+ "," " "* <operand> ")"
<policy> ::= <operator>
```

Test against non-terminal: <policy>

Test a string here!

```
OR('Org1.admin', AND('Org2.peer', 'Org3.peer'), NOutOf(2, 'Org4.member', 'Org5.member', 'Org6.member'))
```

Fig. A.3 Demo of Backus-Naur Form for Signature Policies

Notice how Signature Policies explicitly refer to organizations and their roles. This would make channel configuration cumbersome, as every time a new organization joins or an organization leaves the channel, some or all policies have to be updated. As mentioned earlier in this section, `ImplicitMeta` policies are an alternative to Signature policies aiming to resolve this issue. These policies specify whether `Any`, `All` or `Majority` of organizations are required to perform certain actions. This keyword is followed by a group term such as `Readers`, `Writers`, `Admins`, or `Endorsement`. All organizations map these groups to their own signature policy. As an example, `Org1` may have `Readers` mapped to `Org1.member`, indicating all certificates by that organization are allowed to read according to the organization. A tree-like structure is traversed during the evaluation to find these mappings for the organizations[3].

The grammar of the much simpler ImplicitMeta policies is given below.

[3]Feel free to experiment with this syntax as well: https://bit.ly/3qN6PWZ

$\langle$*quantity*$\rangle$	::= "ANY" | "ALL" | "MAJORITY"
$\langle$*customSubPolicy*$\rangle$	::= ([a-z] | [A-Z])+
$\langle$*subpolicy*$\rangle$	::= "Readers" | "Writers" | "Admins" | "Endorsement" | $\langle$*customSubPolicy*$\rangle$
$\langle$*policy*$\rangle$	::= $\langle$*quantity*$\rangle$ " "+ $\langle$*subpolicy*$\rangle$

Appendix B

GitHub Issues created for Open Issues

B.1 Hyperledger Fabric

1. Extracting functionality from clients to peers to make them thinner: https://jira.hyperledger.org/browse/FAB-18495
2. Requesting historic ledger state by tracking old value on the ledger with rollbacks: https://jira.hyperledger.org/browse/FAB-18496
3. Integrity consideration for small PDCs and transfer-only PDCs: https://jira.hyperledger.org/browse/FAB-18497
4. Chaincode approval is only on metadata, not chaincode implementation: https://jira.hyperledger.org/browse/FAB-18497

B.2 Hyperledger Besu

1. Switching Consensus Protocols: https://github.com/hyperledger/besu/issues/2437

B.2.1 ConsenSys Orion

1. Encountering memory issues related to Toweni/LibSodium: https://github.com/ConsenSys/orion/issues/425
2. Lack of missing data pulling and related issues: https://github.com/ConsenSys/orion/issues/426
3. Risk of private data leaks due to lack of access control: https://github.com/ConsenSys/orion/issues/427

B.2.2 ConsenSys EthSigner

1. Improving fault-tolerance by adding a failover node: https://github.com/ConsenSys/ethsigner/issues/378

B.3 Hyperledger Caliper

1. Warm-up period for benchmarks without waiting time: https://github.com/hyperledger/caliper/issues/1147
2. (PR) Resolve issue in which only first round would be docker-monitored: https://github.com/hyperledger/caliper/pull/1135
3. (PR) Allow specification of custom EOA's and nonces: https://github.com/hyperledger/caliper/pull/1138

B.4 Web3.JS

1. Inefficiency in submitting multiple transactions in bulk: https://github.com/ChainSafe/web3.js/issues/4110

Appendix C

List of Illustrations

C.1 Figures

C.2 Tables

C.3 Listings