Towards a New Generation of Permissioned Blockchain Systems

by

Christian Gorenflo

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2020

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

Supervisors:	Lukasz Golab Associate Professor, Dept. of Management Sciences, University of Waterloo
	Srinivasan Keshav Professor, School of Computer Science, University of Waterloo
Internal Members:	Bernard Wong Associate Professor, School of Computer Science, University of Waterloo
	Jimmy Lin Professor, School of Computer Science, University of Waterloo
Internal-External Member:	Anwar Hasan Professor, Dept. of Electrical and Computer Engineering, University of Waterloo
External Examiner:	Amr El Abbadi Professor, Dept. of Computer Science, University of California, Santa Barbara

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

With the release of Satoshi Nakamoto's Bitcoin system in 2008 a new decentralized computation paradigm, known as blockchain, was born. Bitcoin promised a trading network for virtual coins, publicly available for anyone to participate in but owned by nobody. Any participant could propose a transaction and a lottery mechanism decided in which order these transactions would be recorded in a ledger with an elegant mechanism to prevent double spending. The remarkable achievement of Nakamoto's protocol was that participants did not have to trust each other to behave correctly for it to work. As long as more than half of the network participants adhered to the correct code, the recorded transactions on the ledger would both be valid and immutable.

Ethereum, as the next major blockchain to appear, improved on the initial idea by introducing smart contracts, which are decentralized Turing-complete stored procedures, thus making blockchain technology interesting for the enterprise setting. However, its intrinsically public data and prohibitive energy costs needed to be overcome. This gave rise to a new type of systems called permissioned blockchains. With these, access to the ledger is restricted and trust assumptions about malicious behaviour have been weakened, allowing more efficient consensus mechanisms to find a global order of transactions. One of the most popular representatives of this kind of blockchain is Hyperledger Fabric. While it is much faster and more energy efficient than permissionless blockchains, it has to compete with conventional distributed databases in the enterprise sector.

This thesis aims to mitigate Fabric's three major shortcomings. First, compared to conventional database systems, it is still far too slow. This thesis shows how the performance can be increased by a factor of seven by redesigning the transaction processing pipeline and introducing more efficient data structures. Second, we present a novel solution to Fabric's intrinsic problem of a low throughput for workloads with transactions that access the same data. This is achieved by analyzing the dependencies of transactions and selectively re-executing transactions when a conflict is detected. Third, this thesis tackles the preservation of private data. Even though access to the blockchain as a whole can be restricted, in a setting where multiple enterprises collaborate this is not sufficient to protect sensitive proprietary data. Thus, this thesis introduces a new privacy-preserving blockchain protocol based on network sharding and targeted data dissemination. It also introduces an additional layer of abstraction for the creation of transactions and interaction with data on the blockchain. This allows developers to write applications without the need for low-level knowledge of the internal data structure of the blockchain system. In summary, this thesis addresses the shortcomings of the current generation of permission blockchain systems.

Acknowledgements

First and foremost, I want to thank my supervisors Lukasz Golab and Srinivasan Keshav for supporting and challenging me over the years. You have pushed me to constantly improve myself and my work as well as helped me through difficult periods. I thank Lukasz for his enthusiasm for all our projects and convincing me to come to Waterloo in the first place. I thank Keshav for guiding me to develop a structured approach to both thinking and writing about research ideas and problems.

I thank my committee members Bernard Wong, Jimmy Lin, Anwar Hasan and Amr El Abbadi for their questions and comments which resulted in an improved thesis. In particular, I want to thank Jimmy Lin for pushing back on my initial research direction, which helped sharpen the focus of my work.

I thank Stephen Lee for his collaboration on FastFabric. Our discussions helped me to develop a better understanding of Fabric and taught me how to set up, operate and experiment with such a complex system. I thank the members of the security research group at IBM Zurich for giving me the opportunity to look behind the curtain, work closely with the original creators of Fabric, and expand my research horizon into the field of privacy and security. This includes in alphabetical order, but is not limited to, Elli Androulaki, Marcus Brandenburger, Pasquale Convertini, Angelo De Caro, Gero Dittmann, Kaoutar Elkhiyaoui, Jens Jelitto, Andrea Mambretti, Matej Pavlovic, Miguel Angel Prada Delgado, Alessandro Sorniotti, Chrysoula Stathakopoulou and Marko Vukolić.

I thank my colleagues in the ISS4E and Sirius research groups at the University of Waterloo for making our lab an enjoyable environment for work, study and life in general. In this, I thank Adedamola Adepetu, Michael Doroshenko, Come Carquex, Yerbol Aussat, Ansis Rosmanis, Dimcho Karakashev, Amelia Holcomb, Qingnan Duan, Linguan Yang and Rishav Agarwal. In particular, I thank Costin Ograda-Bratu for going above and beyond whenever his assistance was needed and Fiodar Kazhamiaka as well as Kayla Hardie for not just being colleagues but good friends throughout these years.

I thank my family for their support and encouragement. Even though we were thousands of kilometers apart, I knew you by my side every step of the way. I thank my cousin Frederike for proofreading this thesis. I thank my friends for making my personal life a cherished experience during these years as well. I especially thank Priyank, Camila, Sajin, Tom and Adrián for spending many hours together delving into the deepest dungeons.

Finally, I thank NSERC, SWTCH and the Cheriton School of Computer Science for funding my research.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

One of the long term goals of Computer Science is the creation of systems for reliable computation. This can be achieved in two ways. Faults can either be prevented before they happen or the system needs a way to recover after a fault has occurred. This is especially important in distributed systems. Research into fault tolerant distributed systems reaches back to the 1980s [68, 104, 105]. With the invention of local area networks and later the Internet, it became possible for applications to communicate across hardware boundaries. With this, network failures became a new problem that had to be taken into account when developing a system. Network partitions, i.e. the complete cut-off of one part of the network from another, are commonly considered inevitable [26].

Besides network partitions, single nodes of the distributed application can encounter crash-faults due to malfunctioning hardware or software bugs. This means any data stored on such a node or any service it provides is either unavailable until the node is replaced or it must be redundant in the network. *State machine replication* [65, 67] describes a framework for the management of such network redundancies. It models any application in terms of inputs and modifications to its internal state, making it easier to reason about correct replication across multiple hardware devices. More importantly, faulty replicas must be detectable to either repair or delete them from the system. While exploring this, the fault model was extended to incorporate arbitrary, including malicious, behaviour of replicas, so-called Byzantine behaviour after the Byzantine Generals problem, coined by Leslie Lamport [69]. In the problem, a group of generals has surrounded a city and must decide to whether attack or retreat. Any decision that is not carried out in unison will lead to a catastrophic losses. However, there are some traitors among the generals who can selectively lie to prevent a consensus on the decision. Furthermore, the generals built their camps far apart and need to communicate via messengers, who can in turn forge

or lose messages between generals. It has been shown that more than two thirds of the participants in this consensus need to be well-behaved to achieve a decision [88].

However, for decades malicious behaviour in distributed systems has been mostly an academic problem. In practice, distributed systems were typically owned by single organizations, which made Byzantine behaviour highly improbable. Then, in 2008 a new paradigm shift was heralded by Nakamoto's paper [79] who proposed the *blockchain* technology. While still aligning with the principles of state machine replication, blockchains operate under the assumption that malicious behaviour is not only possible, but probable. Therefore, no single node of the network can be seen as trustworthy. This situation is further exacerbated by the strong notion of decentralization, shifting ownership from a single organization to a multitude of possibly anonymous node owners.

Blockchains manage to operate in such an environment by addressing two problems: they achieve a fault tolerant consensus on decisions concerning all nodes and they make all processed data virtually immutable by making any tampering easily detectable. Consequently, as long as the consensus result is trustworthy, all nodes can reach the same internal state independently and can validate each other's state.

While Bitcoin has been designed for the sole purpose of handling *cryptocurrency*, transferring digital coin from one owner to another, Ethereum introduced smart contracts [117]. With the help of these replicated multi-purpose computations, blockchain technology can truly be seen as a new kind of state machine replication mechanism.

However, a fully decentralized, open and trustless blockchain network comes with drawbacks such as low performance and high computational overhead [73]. They are also very difficult to upgrade and data is completely openly accessible. These properties make such blockchains incompatible with many enterprise settings. Hyperledger Fabric [3] promises to bridge this gap. It slightly alters its trust assumptions, so that all network nodes are known to a consortium of administrators, who can grant or restrict access to the network. This way, only identified and vetted nodes can join the blockchain protocol, keeping a closer control of data access. Furthermore, all network nodes trust a subset of nodes to execute smart contracts correctly, hence reducing the computational overhead and allowing the concurrent execution of different smart contract on these trusted nodes. With the benefit of known participation, Fabric can rely on faster consensus algorithms [116]. This way, it improves on Bitcoin's performance by a factor over 200 and Ethereum by a factor of 10-20 [31]. Still, it falls short of its promise as the enterprise blockchain, because it is too slow to compete with the throughput of conventional distributed databases in existing use cases [102] and, even for new applications that are uniquely enabled by blockchain technology, it is missing crucial features. We first present a brief summary of the description

of Hyperledger Fabric, deferring details to Chapter 2, then we discuss three applications that rely on blockchain technology but which are currently not realizable due to its shortcomings. Subsequently, we give the outline of the thesis, where we state three challenges facing Fabric.

1.1 Hyperledger Fabric

Blockchains are essentially comprised of two data structures, a *ledger* and a *world state*. In the context of state machines, the ledger contains the ordered history of inputs to the state transition function, i.e. transactions, and the world state represents the current state of the machine. Hyperledger Fabric's world state is implemented as a versioned key-value store. A Fabric blockchain network consists of *peer* nodes, which store the ledger and world state, and *ordering* nodes, which order transactions, create blocks and disseminate them to the peers. *Endorsing* peers execute transactions and create a *read* and *write set* of key-value pairs which are used to transition the world state when the transaction is committed. This endorsement is done before the ordering service receives the transaction, so the read-write set is ordered together with its corresponding transaction. After receiving a new block, peers go through every read-write set to check if it is still valid or if the proposed state change has gone stale, i.e. the version number of any of the keys in the set differs from the version number in the state database. Based on this validation, peers either commit or discard any changes to the world state, hence all peers must reach a deterministic conclusion.

1.2 Use cases

1.2.1 Decentralized global payment system

It is widely believed that blockchains will inevitably impact the financial sector. Experts estimate that 10% of the global gross domestic product is stored on blockchains [34]. Here, we discuss one possible application as a global payment system.

Global bank transactions are notoriously slow, because of the fractured banking sector and negotiation overhead. Commonly, transactions take several business days to be cleared [44]. With credit card payments, there is a trusted facilitator such as Visa or Mastercard in the loop. With blockchain technology, we theoretically have the opportunity to build the foundation of a decentralized real-time global payment system that

does not rely on banks or other financial operators. However, in order to be viable in practice, blockchains must support transaction rates comparable to those supported by existing database management systems, which can provide some of the same transactional guarantees. This means blockchains must come close to processing 50,000 transactions per second as required by a credit card company such as Visa during peak times [116].

1.2.2 Democratized marketplace

Today's digital marketplaces rely on large organizations like Amazon or eBay hosting them. These enterprises can act as gate keepers to the market and dictate conditions of use, thereby giving them quasi-monopolies over these marketplaces [46], which has brought on anti-trust inquiries [99, 78]. Effectively, customers and merchants alike have to put their trust completely into the hosting organization. A blockchain platform collectively owned by a multitude of different parties would prevent such dependency on a single organization.

The situation becomes even more complicated when the traded goods are completely digital. A prominent case is the trade with carbon emissions, which is plagued by fraud [83]. While each certificate should represent a certain amount of emissions, in reality we see that conventional data base management systems have difficulties preventing double-spending. Blockchains on the other hand, excel at tracking scarce assets, be it *digital anchors* for real assets or completely digital ones.

However, taking a renewable energy trading platform as an example, we find blockchains to suffer from a different set of problems than their conventional counterparts. Similar to the case of the global payment system they need to be able to support a certain transaction throughput. Currently only blockchain systems like Hyperledger Fabric, which allow parallelized execution of transactions, can come close to achieving this. What is more, a marketplace produces a mapping between consumers and producers of energy. Generally, there will be more consumers than producers, because large utilities aggregate much of the production. As a consequence, many consumers try to buy renewable energy from the same producer. Because of Fabric's parallelized smart contract execution model, contention in the workload leads to many conflicting transactions. Hence, the overall transaction throughput suffers significantly. This bottleneck has to be removed for blockchains to become a true alternative to marketplaces owned by single organizations.

1.2.3 Collaborative supply chain

In the logistic sector, goods go through many hands before they reach their final destination. With conventional databases this means that either each partner manages their own data silo or one partner owns all the data and all others use their system. The latter case would lead to a lot of accumulated power in the hands of the hosting organization similar to the marketplace case. In the former case, tracking of goods along the supply chain can be difficult, data can get lost or is not available for the necessary parties to access. For example, an outbreak of E.coli infections in the US in November 2019 took weeks to trace back to the source [20]. Even then, it was only possible to localize it to a whole region, because finer-grained tracking mechanisms were missing. IBMs FoodTrust initiative based on Hyperledger Fabric aims to tackle exactly this problem, by tracking food sources on a blockchain [54]. However, Fabric's privacy settings are limited and leak information to competitors even when all the data on the ledger is encrypted. This is even more problematic in a project like TradeLens [55], where hundreds of different events surrounding the shipment of goods can be tracked. Each of these events can have different visibility along the supply chain. For example, the loading of a shipment into a container might be visible for the involved port and ocean carrier, as well as the manufacturer and final customer, but not to any other transport operator later down the line. Such complex layered privacy settings are currently only possible with IBM as a gate keeper in the loop. But this reduces the system back to a trusted party system. Blockchain systems need to be able to handle both collaborative sharing of data as well as dynamic fine-grained access control to live up to its promise.

1.3 Thesis Outline

This thesis aims to mitigate the shortcomings of blockchain systems for large scale enterprise applications. We base our research on Hyperledger Fabric, because of its modularity [51] and comparably high throughput [31] we believe it is the most promising candidate to date.

We discuss the necessary background for this work in Chapter 2. This is followed by the three central chapters of the thesis. In Chapter 3 we describe our work on general performance optimization of Hyperledger Fabric, while we look into more specialized workload based optimizations in Chapter 4. Then, we examine and improve the privacy and usability aspects of Fabric in Chapter 5. We collect our findings in Chapter 6 and point out potential future work. In the following, we outline the contributions of the three central

chapters.

1.3.1 Contributions

Before we go into a more detailed description of the contributions of each chapter, we want to motivate the importance of this work at a high level. The uniquely defining quality of blockchain technology is that it replaces trusted entities and can give reliable decentralized access to both data and computation. Conversely, when vying for adoption, it directly competes with existing systems based on trusted entities. However, in comparison with these legacy systems current blockchains suffer from three primary limitations: lack of performance, lack of data privacy, and coupling of computations with data, as described next.

So far, existing blockchain systems are sorely lacking in performance. This means that large-scale applications either still need to rely on high-performance trusted entities (see Section 1.2) or they are simply not feasible to implement at all if no such entities are available.

The next issue that needs to be resolved is the lack of data privacy. By design, data on a blockchain is shared between a multitude of network nodes. Especially in the enterprise environment, sharing valuable internal data with competitors is undesirable. To address this, some current blockchain systems implement additional data privacy features on top of their basic functionality. These features suffer from one or more of these three deficiencies: They are insufficient and hence can leak data; they introduce a trusted gate-keeper for data access, thereby negating the reason to use a blockchain; or they sacrifice performance by relying on complex cryptographic computations such as zero-knowledge proofs.

Lastly, current blockchains tightly couple computation with the data on the ledger. Usually, smart contracts define precisely how data is stored, meaning there is no data layer abstraction between business logic and data storage. Moreover, they describe interactions such as bank transfers. Consequently, for every type of interaction that needs to be enabled on the ledger, a new smart contract must be written and becomes fixed once committed to the ledger. As application requirements change over time, this impedes their ability to evolve the corresponding business code. Lessons learned from other fields such as system architecture and programming languages need to be applied to blockchain design to keep blockchain systems manageable and flexible for the entire lifetime of the application. Without advancements in smart contract and data storage design, competing systems such as distributed databases provide superior usability with clearly defined application interfaces

that decouple internal processes from outside consumers. With the improvements in performance, privacy and usability proposed in this thesis, blockchain systems will be able to compete with legacy systems based on centralized trusted parties and move from a niche technology to a serious choice for mainstream enterprise applications. In the following, we describe how each chapter of the thesis tackles these challenges.

Performance

Although the consensus algorithm has usually been the bottleneck for blockchain systems, recent work [108, 60, 120, 110] has started to address its shortcomings, motivating us to look beyond consensus to identify further performance improvements.

In Chapter 3 of this thesis, we critically examine the design of Hyperledger Fabric other than the consensus protocol from a raw performance perspective. While there has been some work on optimizing Hyperledger Fabric, e.g., using aggressive caching [113], there has been no attempt to rearchitect the system as a whole. Hence, we design and implement several architectural optimizations based on common system design techniques that together improve the end-to-end transaction throughput by a factor of almost 7, from 3,000 to 20,000 transactions per second, while decreasing block latency. We achieve this by breaking the assumption that blockchain nodes must be single units of hardware. We decouple parts of the blockchain code to distribute it to multiple servers and make more efficient use of available computational resources. Our specific contributions on improving Fabric's performance are as follows:

1. *Separating metadata from data:* the consensus layer in Fabric receives whole transactions as input, but only the transaction IDs are required to decide the transaction order. We redesign Fabric's transaction ordering service to work with only the transaction IDs, resulting in greatly increased throughput.

2. *Parallelism and caching:* some aspects of transaction validation can be parallelized while others can benefit from caching transaction data. We redesign Fabric's transaction validation service by aggressively caching unmarshaled blocks at the committers and by parallelizing as many validation steps as possible, including endorsement policy validation and syntactic verification.

3. *Exploiting the memory hierarchy for fast data access on the critical path*: Fabric's key-value store that maintains world state can be replaced with light-weight in-memory data structures whose lack of durability guarantees can be compensated

by the blockchain itself. We redesign Fabric's data management layer around a lightweight hash table that provides faster access to the data on the critical transaction-validation path, deferring storage of immutable blocks to a write-optimized storage cluster.

4. *Resource separation:* the peer roles of committer and endorser vie for resources. We introduce an architecture that moves these roles to separate hardware.

Importantly, our optimizations do not violate any APIs or modularity boundaries of Fabric, and therefore they can be incorporated into future releases of Fabric [52][1]. This work was published in ICBC 2019 and a special issue of the International Journal of Network Management (extended version) [42, 43].

Skewed workloads

Uncoordinated execution of smart contracts in a decentralized network can result in inconsistent blockchains, a fatal flaw. Fundamentally, blockchain systems have two options to resolve such conflicts. They can either coordinate, i.e., execute contracts after establishing consensus on a linear ordering, or they can deterministically resolve inconsistencies after parallel execution.

Most existing blockchain systems implement smart contract execution after ordering transactions, giving this pattern the name *order-execute* (OX). In these systems, smart contract execution happens sequentially. This allows each execution to act on the result of the previous execution, but restricts the computation to a single thread, limiting performance. Blockchains using this pattern must additionally guarantee that the smart contract execution reaches the same result on every node in the network that replicates the chain, typically by requiring smart contracts to be written in a domain-specific deterministic programming language. This restricts programmability. Moreover, it makes the use of external data sources, so-called *oracles*, difficult, because they cannot be directly controlled and may deliver different data to different nodes in the network.

Other blockchain systems, most notably Hyperledger Fabric, use an *execute-order* (XO) pattern. Here, smart contracts referred to by transactions are executed in parallel in a container before the ordering phase. Subsequently, only the results of these computations are ordered and put into the blockchain. Parallelized smart contract execution enables, among other benefits, a nominal transaction throughput orders of magnitude higher than

[1]Source code available at https://github.com/cgorenflo/fabric/tree/fastfabric-1.4

that of other blockchains [31]. However, a model that executes each transaction in parallel is inherently unable to detect transaction conflicts during execution. Two transactions are said to conflict if either one reads or writes to a key that is written to by the other.

Prior work on contentious workloads in Fabric focuses on detecting conflicting transactions during ordering and aborting them early. However, this tightly couples architecturally distinct parts of the Fabric network, breaking its modular structure. Furthermore, early abort only treats a symptom and not the cause in that it only filters out conflicting transactions instead of preventing their execution in the first place. This approach will not help if many transactions try to modify a small number of 'hot' keys. For example, suppose the system supports a throughput of 1000 transactions per second. Additionally, suppose 20 transactions try to access the same key in each block of 100 transactions. Then, only one of the 20 transactions will be valid and the rest must be aborted early. Subsequently, all 19 aborted clients will attempt to re-execute their transactions, adding to the 20 new conflicting transactions in the next block. This leads to 38 aborted transactions in the next round, and so on. Clearly, with *cumulative re-execution*, the number of aborted transactions grows linearly until it surpasses the throughput of the system. Thus, if clients re-execute aborted transactions, their default behaviour, this effectively becomes an unintentional denial of service attack on the blockchain!

This inherent problem with the XO pattern greatly reduces the performance of uncoordinated marketplaces or auctions. For example, conflicting transactions cannot be avoided in use cases such as payroll, where an employer transfers credits to a large number of employees periodically, or energy trading, where a small number of producers offer fungible units of energy to a large group of consumers.

Our solution XOX Fabric essentially adds a second deterministic re-execution phase to Fabric. This phase executes 'patch-up code' that must be added to a smart contract. We show that, in many cases, this eliminates the need to re-submit and re-execute conflicting transactions. By analyzing the dependency structure of transactions executed in the pre-order execution step, we add the benefit of transaction conflict resolution to the XO model while minimizing the additional computational overhead. Our approach can deal with highly skewed contentious workloads with a handful of hot keys, while still retaining the decoupled, modular structure of Fabric. Our contributions described in Chapter 4 are as follows:

- *Hybrid execution model:* Our *execute-order-execute* (XOX) model allows us to choose an optimal trade-off between concurrent high-performance execution and consistent linear execution, preventing the cumulative re-execution problem.

- *Compatibility with external oracles:* To allow the use of external oracles in the deterministic second execution phase, we gather and save oracle inputs in the pre-order execution step.

- *Concurrent transaction processing:* By computing a DAG of transaction dependencies in the post-order phase, Fabric peers can maximize parallel transaction processing. Specifically, they not only parallelize transaction validation and commitment, making full use of modern multi-core CPUs, but also re-execute transactions in parallel as long as these transactions are not dependent on each other. This alleviates the execution bottleneck of OX blockchains.

We achieve these contributions while being fully legacy-compatible and without affecting Fabric's modularity. In fact, XOX can replace an existing Fabric network setup by changing only the Fabric binaries[2]. This work is going to be published in ICBC 2020.

Privacy & usability

Blockchain systems promise to store an immutable ledger of ordered transactions. The execution of these transactions creates the blockchain's world state. Hyperledger Fabric creates and stores the world state explicitly as a by-product of the execution of smart contracts. World state is typically replicated across all nodes in the blockchain network.

By design, the replication of the world state is at odds with privacy. In particular, in enterprise use cases companies might want to share only some parts of their data with a business partner. The common approach to this problem is to store only hashes of sensitive data on the ledger and keep the actual data on a separate storage device [58]. However, this approach has its limitations as smart contracts can only operate on data available to a node.

Alternative proposals that base privacy on cryptographic primitives such as zero knowledge proofs (e.g., [9]) are usually complex and computationally expensive and therefore not widely applicable.

In Fabric, clients can define a subset of peers to be trusted with private data. Only this trusted set sees the results of the clients' transactions in clear text. However, even untrusted peers have to come to a deterministic conclusion about the validity of a transaction. Because the key names and version numbers are necessary to decide transaction validity, all peers still have to see them; only the key values can be kept private. This leaks

[2]Source code available at https://github.com/cgorenflo/fabric/tree/xox-1.4

information about which transaction touches which keys to the whole network, including untrusted peers.

Making read-write sets explicit public parts of transactions reveals another problem of Fabric. To write a smart contract, it is not sufficient to be a domain expert in the represented business case. Developers additionally require a good understanding of Fabric's internal structure. At the very least, they have to understand how to interact with Fabric's world state interface. This means they need to translate their business domain into simple *put* and *get* commands to interact with Fabric's internal key-value store. This tightly couples smart contracts to Fabric's internal implementation, so Fabric cannot easily evolve its data structures without breaking existing smart contracts. Moreover, developers need to be aware of Fabric's key validation to understand how transaction conflicts occur and what implications that will have for their application. For example, if the key space is not properly designed, transactions can easily create workloads that make heavy use of a few hot keys, as we discuss in Chapter 4. Furthermore, to make full use of blockchain's immutability and trust guarantees, all business logic surrounding a transaction must be done inside of smart contracts. However, smart contract execution is replicated on many nodes. That means any extra work is amplified by the replication factor. Additionally, smart contracts are practically not composable, which is why every smart contract must perform a desired atomic action in its entirety. But the more logic a smart contract contains the narrower becomes its applicability. In essence, whenever there is a new requirement, a new smart contract has to be written.

In Chapter 5, we address both privacy and ease-of-use issues by designing TNG, a composable privacy-first blockchain framework without relying on expensive cryptographic primitives. This work relies on the key insight that current blockchains treat trusted data storage as an add-on and reuse their ill-fitting mechanisms for fault-tolerant computation to provide some data privacy. We create a model which addresses data privacy first and builds fault-tolerant computation and immutability on top of it. With this framework we make the following contributions:

- *Shard-based blockchain framework:* We use sharding not only for increased parallel execution, but also to implement targeted dissemination of data. Our novel solution allows fast atomic inter-shard transactions without leaking information to curious observers.

- *Privacy-preserving non-blocking atomic commitment protocol:* Because transaction validation spans multiple shards with distinct views of the world state, an atomic commit protocol must be put into place to ensure cross-shard consistency. We propose

a novel algorithm that allows shards to agree on a result without learning anything about the state change in other shards. Furthermore, we prove the consistency of the protocol even when shards only learn of a sparse subset of other involved shards.

- *Composable domain-driven transaction model:* Instead of using smart contracts to interact with the world state, we introduce *smart assets*, data representations that encapsulate and manage their own internal state and which can be manipulated through function calls. Applications can make use of these smart assets without knowledge of the underlying blockchain system. Arbitrarily many calls to smart asset functions can be composed into a single transaction to tie their executions atomically together.

At the time of writing this work is a pending submission to ACM PODC 2020.

Chapter 2

Background

In this chapter, we describe in detail the pieces that are needed to create a blockchain system. For a more rigorous approach, we first recast important definitions of the blockchain space in terms of the theoretical model of state machine replication. Then, we describe implementation specific characteristics that arise from its intrinsic philosophy. Lastly, we present three major blockchain systems.

2.1 State machine replication

A State Machine is described by the tuple $(Q, \Sigma, \delta, q_0, F)$.

- Q is the set of possible states
- Σ is the alphabet of inputs
- δ is the state transition function
- q_0 is the starting state
- F is the set of final states

For any given input and current state the machine is in, the state transition function returns the new state of the machine after the transition. Assuming there is a chance that the state machine can be faulty, a possible way to prevent or at least tolerate failures is to

create redundant replicas. For better fault-tolerance, these replicas should be executed on independent hardware.

In the beginning, every replica starts in the same state q_0 and has the same state transition function δ. At any given point in time we can check for inconsistencies between replicas, if the following two requirements hold:

- Every replica receives the same set of inputs
- The input order is identical for all replicas

If this holds true, all replicas must be in the same state q^* after processing k inputs. Otherwise, even well-behaved replicas could deviate simply by receiving a different input. Given identical input, faulty replicas can be tolerated by exchanging the current states among replicas and comparing them. Here, we must differentiate two failure models:

Fail-stop failure A machine incurred a failure that puts it into a faulty state, but is otherwise well-behaved. In particular, it responds to queries from other replicas honestly and stops once the failure is detected. In this case, f failures can be tolerated by communication among $2f + 1$ replicas [104].

Byzantine failure A machine can behave arbitrarily and even lie about its own state. Because it can send different responses to different replicas, it can make well-behaved replicas look faulty by reporting an untrue state. In this case at least $3f + 1$ replicas are needed to tolerate f faulty ones [88, 69].

In the following, we define blockchain terms using this model. Note that only a fault in the system leads to a final state, otherwise blockchain systems are designed to run for an indefinite amount of time.

Node A blockchain node is the (virtual) server that runs one instance of the state machine. The state machine is given by the source code of the blockchain system. It ensures that the history of state transitions based on its input is *linearizable* [47]. A history of events, e.g. state transitions, is linearizable if and only if the history H of their invocations and finalizations is equivalent to a valid sequential history S and the partial order of H is a subset of the partial order of S. Effectively, this means that for the sake of validity it can be assumed that state transitions happen instantaneously.

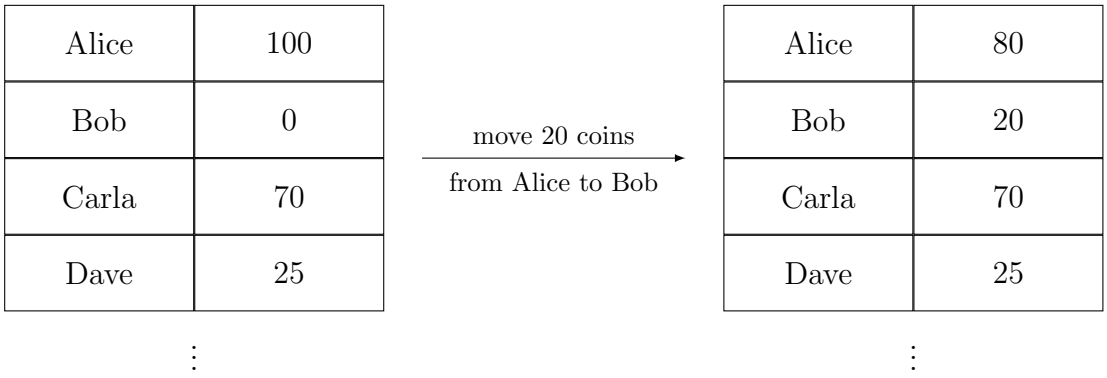


Figure 2.1: Partial world state transformation by a transaction.

Ordering service The ordering service ensures that the order of all inputs to the nodes is fixed. Together with the linearizability of state transitions, this guarantees deterministic equal outcomes on each node.

Network All replication nodes together with the ordering service form a blockchain network.

Transaction A transaction is an instance of an input to the blockchain state machine.

Client A client is some entity that creates transactions and sends them to the network. Additionally, clients can query nodes for information on their state.

World state The world state is the current state of any well-behaved replica. It itself consists of many sub-states. Generally, the world state after modification by an input transaction is mostly the same apart from some isolated sub-states which were impacted by the transaction. See Figure 2.1 for an example of a world state transformation.

Ledger The ledger is the ordered history of all past transactions received by the network. While this usually takes the form of a list, is not necessarily so. As a generalization, it can be represented by a directed acyclic graph (DAG).

Genesis block The genesis block is used to bootstrap each replica and acts as the starting state of the state machine.

Smart contracts Smart contracts are small executable programs that take in a transaction as input and create an output that modifies the world state of the blockchain. Therefore, they form the state transition function.

Asset An asset is a an encapsulation of properties that represents either a virtual or real entity. The execution of a smart contract either deletes or modifies a property of an existing asset or creates a new one. The world state comprises all currently available assets. Common examples are coins of a cryptocurrency or the digital anchor of a real asset like a house.

With these building blocks we have defined the theoretical structure that forms the foundation of a blockchain system. However, the characterizing properties that are layered on top of this are *decentralization*, a *trustless* environment and *immutability*. In the following we will discuss how each of these properties is achieved.

2.2 Permissioned vs. Permissionless blockchains

A replicated state machine by its very nature is a distributed system. However, from its conception [79] blockchains are not only distributed but *decentralized*. This means that there is no explicit hierarchy between nodes of a blockchain network. In fact, new nodes might join the network and existing ones might leave. Overall functionality should not be impeded as long as sufficiently many nodes carry the world state over during such a transition.

This gives rise to the question of network membership. In its purest form, a blockchain system allows any node to join the network, as long as it runs compatible code and adheres to the protocol in place. Such a system is called *permissionless*, because there is no governance model in place which controls which nodes belong to the network. Here, network membership is an emergent property: New nodes are bootstrapped with the addresses of known network nodes to signal their addition to the network. This information is then propagated through the network until at least a large enough number of nodes have knowledge of the change. Likewise, if a node is unresponsive to requests from other nodes, they will remove it from their internal view of network membership. Permissionless blockchain technology is heralded by its proponents as a truly democratic system with equal rights and opportunities for all and in which the code itself is law [28]. However, a decentralized system without any ownership and without built-in governance is also almost impossible to steer in a certain direction. Updates are very difficult to implement, because every node decides independently which code it runs. Therefore, updates must be either non-breaking, so nodes can adopt them gradually, or the majority of nodes must decide to switch together at the same time. This prevents adjustments to counteract unintended developments like

the formation of large mining pools in Bitcoin [70] or countermeasures against network attacks [103].

To mitigate these problems, *permissioned* blockchains give up total decentralization for enhanced administrative power. These networks keep track of node memberships and have gated access. Instead of each node storing their own membership view, they install a membership provider that has the final say. Generally, network membership is under the control of a set of administrators. As a consequence, nodes in the network are vetted and their identity is known, which significantly lessens the probability of malicious behaviour. For these reasons, permissioned blockchains are more popular in enterprise collaborations, where it is undesired to give free network access to anybody but the collaborators. As a side effect of having a globally known membership status, more efficient algorithms can be used to come to a consensus among nodes, as we will describe in Section 2.3.1. Furthermore, at any time, the administrators can issue updates to the code, remove misbehaving nodes or even roll back the whole system to a previous state. To prevent the complete loss of decentralization, special care must be taken that the set of administrators is chosen fairly from among all collaborators and policies like majority voting on decisions are put into place.

2.3 Trust

The second key feature of blockchains is that they don't require any specific network node to behave correctly for the whole protocol to work. Specifically, nodes do not need to trust each other when they communicate. The only requirement that blockchain protocols impose is that a certain percentage p of the overall network is well-behaved. The particular value depends on the protocol, typical values are more than half [79] and more than two thirds [69].

We start by assuming an idealized environment where every node has complete information and this information is ordered deterministically. Here, a trustless network is trivially achievable by foregoing communication completely. Every node replicates all necessary processing steps and reaches the same conclusion after x steps, provided the node is well-behaved. For f assumed stop-failures a client simply queries up to $f + 1$ nodes to receive at least one correct response. If instead of being crash-faulty these nodes can exhibit malicious behaviour, clients need to query $2f + 1$ nodes since f responses can be lies. Then they choose the majority answer to receive the correct result.

When information is incomplete, nodes need to communicate. Then, nodes in the network need to agree on a common basis to proceed with the next step. This means they

face the same challenges dealing with faulty nodes as clients did in the previous illustration. This agreement between nodes is regulated by a *consensus* algorithm. While it is still in the node's best interest to only rely on their own computation whenever possible, in reality consensus algorithms are a necessary puzzle piece to make the trustless environment work. We discuss this next.

2.3.1 Consensus

Most commonly, consensus algorithms in blockchains are used to decide the order of the next batch of transactions. Because the network overhead of n-to-n communication between all network nodes is too large, consensus algorithms start by electing a smaller committee or a single leader either temporarily or permanently. Since these nodes are responsible for the ordering of transactions, we call them the ordering service.

In the highly fluctuating environment of permissionless networks, it is not feasible to elect specific nodes for long periods of time. Instead, the most common solution is to start a lottery for each round of consensus and elect the winner(s) to be part of the ordering service. In permissioned blockchains, the ordering service can be permanently set as a network configuration, but even then it proves to be beneficial to switch the leader of the consensus often, should the protocol rely on a leader. In each case it is possible for a malicious node to become the leader of the ordering service. While other nodes can detect fabricated transactions and consequently discard the consensus result, malicious leaders can choose to omit specific transaction to provide an advantage to the ones they want to prioritize. We will now present the most common consensus schemes. Because the ordering service must give proof to the rest of the network that they are indeed eligible to decide on the transaction order, different consensus protocols are typically collected into categories of *Proof of X*. To improve performance, consensus is not only achieved for one transaction at a time, but for batches of transactions, called *blocks*.

Proof of Work

In this approach, the right to propose a new block goes to whichever node can prove that it expended some resources defined by the blockchain implementation. To secure the blockchain against attacks through block flooding, the work must be expensive, but the proof easily verifiable. Moreover, to give any node in the network a fair chance of finding a proof, a probabilistic algorithm is used that gives a chance of success proportional to the node's computational power. The proof is disseminated as part of the block proposal. If

both the block and the proof are valid, they will be distributed by other nodes as well, otherwise they get discarded. Owners of nodes that participate in the search for new blocks are commonly referred to as *miners*. In most PoW instances, the miner of a valid block is compensated in some manner. Due to the algorithm's probabilistic nature, it is possible that multiple nodes propose new valid blocks at the same time, leading to competing transaction histories, called a *fork*. Usually, there is a mechanism in place so that over time one of the two branches gets abandoned, invalidating all transactions in that partial history in the process. Proof of Work is a typical solution for permissionless networks with Bitcoin as its most prominent representative (see Section 2.6). It is generally criticized for its highly wasteful computation [73].

Proof of Stake

While similar to PoW, PoS tries to minimize the computation to reach a valid proof without opening up the network to new attack vectors [17]. In this case, a leader is chosen through a calculation based on the recent transaction history and some stake in the system, such as the coin balance (ownership of cryptocurrency) of the miner in question [84] or the age of their coin, meaning how long the coins were in their possession [61]. The bigger the stake in the system, the more probable it generally is to meet the requirements for a Proof of Stake. Instead of using rewards as an incentive to mine valid blocks, miners put their stake on the line when proposing a new block. If they propose an invalid block, they lose access to at least part of their assets already on the chain. This incentivizes them to cooperate in the operation of the system.

Proof of Authority

In contrast to the previous two consensus variants, in PoA the nodes participating in the consensus are known. This requires some form of external influence over the network structure, therefore it is impossible in a totally permissionless blockchain. However, while the consensus nodes need to be an identified and approved group, they can form a portion of a bigger permissionless network where anyone can create a node to replicate the blockchain state and verify the validity of created blocks. Because all nodes in the permissioned consensus subgroup know the identity of all members, election algorithms become possible, where a valid majority vote is eventually agreed upon. Prime examples for such algorithms are Paxos [66] and Raft [87]. However, these algorithms can only deal with stop-failures, not Byzantine behaviour. As stated before, consensus tolerant to f Byzantine nodes can be achieved if there are at least $3f + 1$ nodes in the consensus group [88], which is easily

demonstrated. Assume there are f Byzantine faulty nodes that can do anything from not responding at all or delaying their response to responding with a false value or even telling the truth. For any node to receive at least one response from a well-behaved node, they would have to contact $f+1$ nodes. But the malicious nodes could all lie about the true value, so a majority vote is needed. This requires $2f + 1$ votes. However, now malicious nodes could boycott the consensus algorithm by not responding at all. To ensure the algorithm can always make progress, at least $2f + 1$ well-behaved nodes are needed, bringing the total number of nodes to at least $3f + 1$. The most widely used practical implementations of a Byzantine fault tolerant (BFT) algorithm are PBFT [19] and Tendermint Core [112].

PoW and PoS can only achieve *eventual* probabilistic consistency due to the possibility of ledger forks. This leads to networks relying on these protocols to be susceptible to so-called 51% attacks [119]. In such a scheme an attacker controls more than 50% of the total mining power or stake respectively. This allows the attacker to mine blocks faster than the rest of the network. Therefore, they can mine a number of blocks in secret, then present them all at once to the network. Consequently, fork of the transaction history is created from when they split off the main ledger. Because disputes are settled by continuing on the ledger which costs the most amount of work, the malicious ledger wins out, erasing all transactions in the abandoned history. The attacker can take advantage of this by spending their digital coins on real-world goods or services with transactions that get erased by their own alternative ledger history. This is called a double-spending attack [59], because after the old history gets abandoned the attacker can spend the same coins on something else.

Because of the danger of double-spending attacks, the probability for accidental forking must be kept low. To this end, PoW even trades performance implicitly for increased security, since an easier search for the proof increases the chance of multiple miners finding one simultaneously. This leads to low transaction throughput and high latency until confirmation of success. Throughput ranges typically between one and a few hundred transactions per second with a latency in the order of minutes to hours [31].

In contrast to that, the *Canopus* algorithm [101], a non-Byzantine-fault-tolerant PoA consensus, showed a throughput of several million transactions per second with latencies in the sub-second range. This is achieved by taking advantage of the topology of the blockchain network, restricting the majority of communication to servers in the same rack or at least the same data center while fetching data from remote nodes only if absolutely necessary. Its Byzantine fault-tolerant extension RCanopus [60] and Mir-BFT [110], as well as ResilientDB [45] demonstrate that the ordering service of a network no longer has to be the bottleneck of the system.

2.4 Immutability

The last of the breakthrough properties of blockchain systems is the immutability of historical data. Data once recorded on the ledger cannot be modified without leaving easily recognizable evidence for tampering. This guarantees that a node will never be able to convince anyone else of the veracity of a tampered ledger. It is especially important because the world state can be reconstructed by replaying all transactions recorded on the ledger. Therefore, inconsistencies in the world state between two nodes can easily be rectified by comparing their respective ledgers. The immutability guarantee relies heavily on cryptographic primitives, which we will discuss in detail in the following sections.

2.4.1 Cryptographic Hashes

A function $H : X \to Y$ that maps deterministically from an input X with $|X| = N$ to a set of output values Y with $|Y| = M$ is considered an (N, M) hash function. Typically $N \gg M$, so that H acts as a lossy compression mechanism. Most prominently, they are used in Computer Science to create fast-lookup data structures like hash tables. They also find applications in the field of cryptography, if they satisfy the following properties and are therefore considered secure [111]:

Pre-image resistance For a given *hash* or *digest* h, it must be difficult to find the original message m, so that $H(m) = h$.

Second pre-image resistance For a given input m, it must be difficult to find a second input m' with $H(m) = H(m')$.

Collision resistance It must be difficult to find two messages m_1 and m_2 with $H(m_1) = H(m_2)$.

Moreover, cryptographic hash functions should behave like (pseudo-)random functions, so that small changes to the input lead to completely different outputs. Generally, this implies that only brute force attacks can be applied to create intentional collisions and the set of possible digests is sufficiently large to prevent accidental collisions. Such primitive (N, M) hash functions can be iteratively applied to an input of arbitrary length [77].

A secure hash can be seen as a kind of digital fingerprint of the input message. This allows opaque identification and authentication of specific data by means of its much shorter hash.

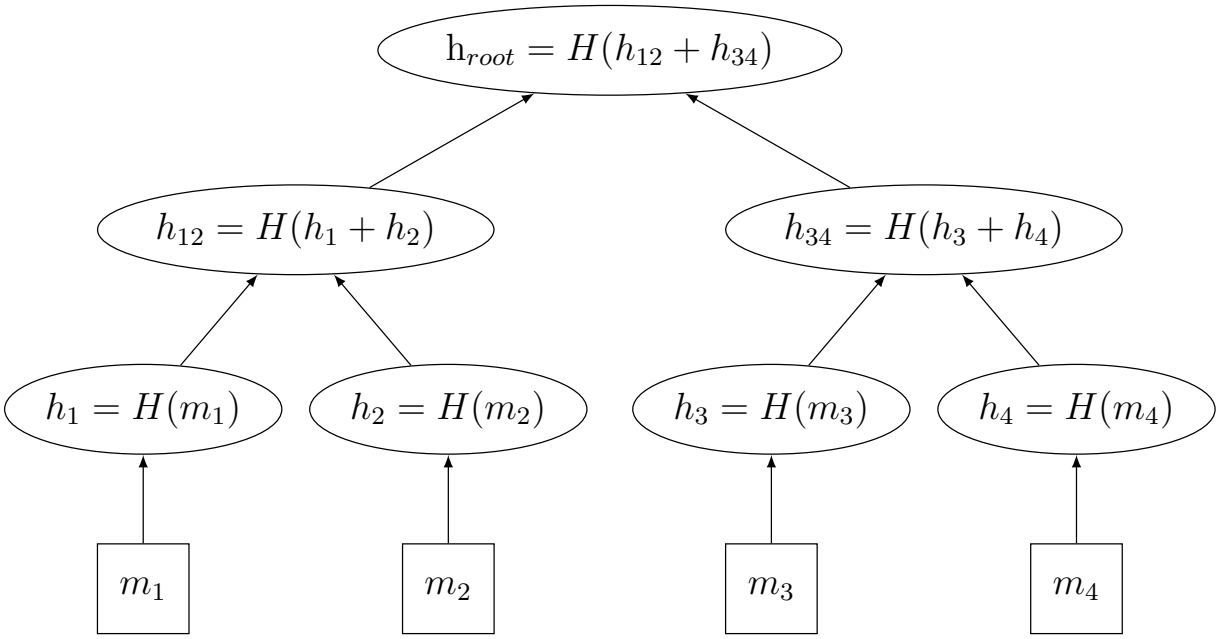


Figure 2.2: Construction of a Merkle Tree. The hash function H takes the initial messages m_i as inputs. Parent nodes are constructed by concatenating child hashes and rehashing them.

2.4.2 Merkle Trees

Because hashes identify the underlying data to a reasonable certainty, they can also be used to build more complex data structures to represent the order of messages. Let $m_1, ..., m_n$ be an ordered list of n messages. Then a Merkle Tree [76] over this list represents a unique identifier for its order. To construct a Merkle Tree, the data blocks get hashed and become the leaf nodes of a tree. Then, two hashes at a time are concatenated and hashed again to form the parent node of the two leaf nodes. This is repeated recursively until a single root node is reached. Figure 2.2 shows the construction for four initial messages.

Should any message change or the order be permuted, then all parent hashes that are affected by the modification would also change. That means that any node of the tree can be used to verify that the data that formed its subtree was not mutated. As a consequence of using hashed messages as its leaf nodes, a Merkle tree can be constructed collaboratively by multiple actors without revealing the full list of messages to any given actor. Using the example of four messages, let actor a_1 know m_1, actor a_2 know m_2 and m_3 and actor $a3$ know m_4. Then, a_1 is able to construct the Merkle Tree by asking a_2 for h_2 and asking

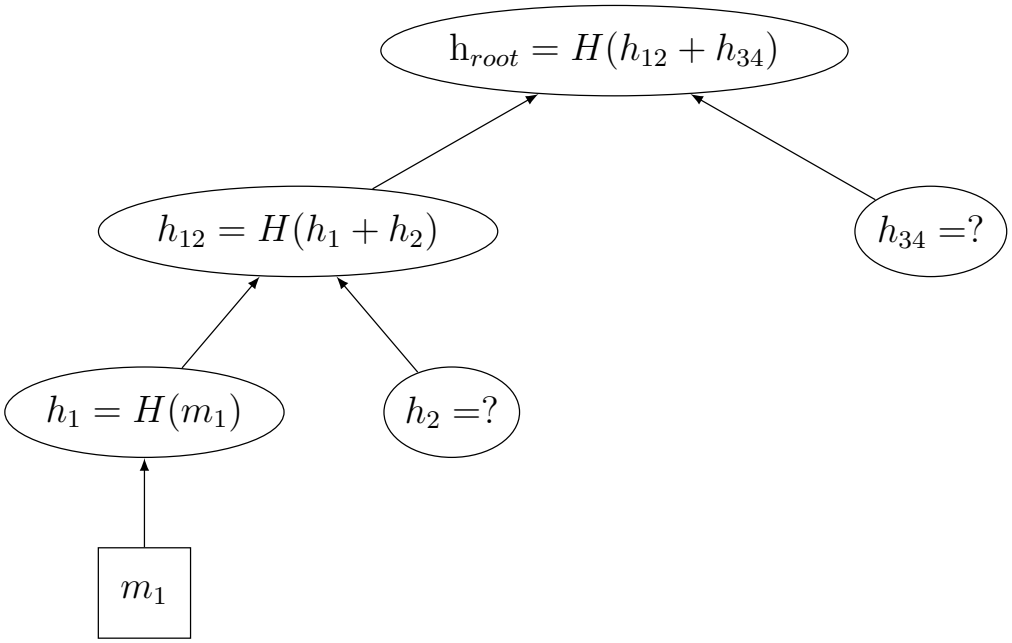


Figure 2.3: Partial view of a Merkle Tree if only m_1 is known initially.

either a_2 or a_3 for h_{34}. The actor a_1 never needs to know the actual content of the original messages to calculate the correct Merkle root as shown in Figure 2.3.

2.4.3 Cryptographic signatures

Blockchains, permissioned ones in particular, make avid use of a public key cryptography [100]. To this end, each actor, be it network node or client, creates a pair of private and public cryptographic keys. While the public key is broadcast to the network, the private key is kept hidden. Now, everyone can use the public key of a specific node to encrypt a message that only this node can decrypt with its private key to prevent eavesdropping. Yet, this mechanism can also be used in reverse. If a private key is used to encrypt data, as long as the key has not been leaked, the owner of that key is the only one to be able to create that cipher, but everyone else can decrypt it with the public key. Therefore, this can represent a digital signature of the sender.

Furthermore, the public key can be coupled to a certificate. Certificate Authorities (CAs) are trusted parties that can create signed certificates. A certificate can comprise information about the sender like name, country and organization as well as a known

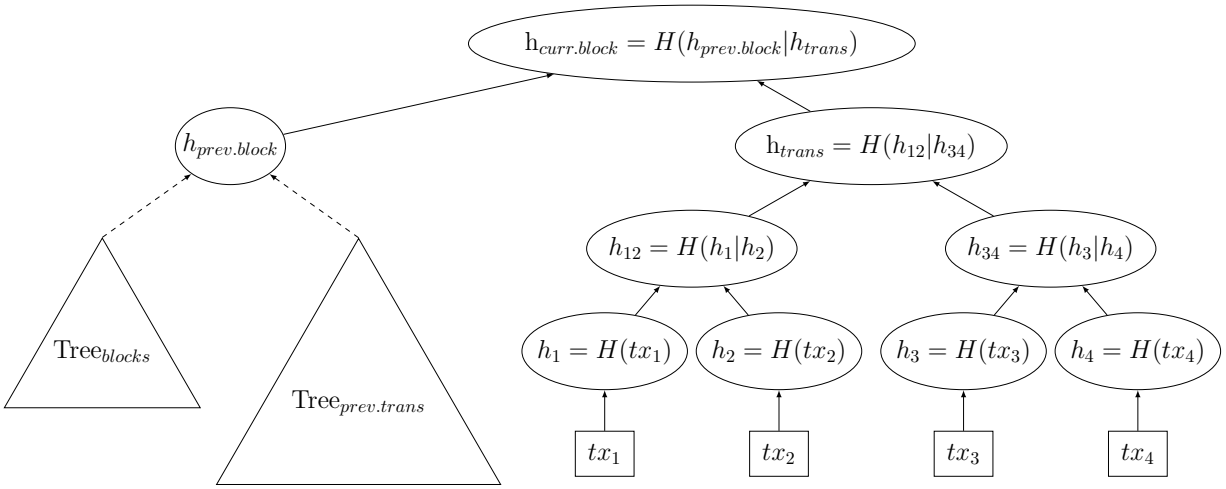


Figure 2.4: Example of a blockchain Merkle tree. Transactions tx_i create a Merkle subtree. Its root hash is then combined with the previous block's hash to create the current block's hash. This algorithm is recursively applied for every new block.

public key. This way, the CA attests that this public key belongs to the holder of the specific certificate. Such certificates are used by permissioned blockchains to identify their members.

Since a hash acts as the fingerprint of the full message, it is enough for a sender to sign the hash to keep the signature as small as possible so the communication overhead stays low.

2.4.4 Immutable ledger

Now all building blocks to create an immutable ledger are in place. Starting with the transactions, care must be taken that they themselves cannot be modified at any time after the original senders emitted them. Therefore, each sender also signs the transaction hash and adds it to its message. Now, this signature permeates the specific hash and anyone can check the transaction data to verify its correctness.

Next, the order of transactions must be fixed. As mentioned in Section 2.3.1, for improved performance transactions are batched into blocks by the ordering service. This means every block has a distinct hash which anchors the specific transaction order it consists of. While a simple hash of the block can be used to detect any tampering, it is too coarse a tool to verify a specific transaction's position. To this end, a Merkle Tree is

used to create a root hash. Now, by revealing the complete structure of the Merkle Tree any modification can be pinpointed. As an additional benefit, this Merkle Tree can grow recursively: The root hash of the current tree forms the left child of the tree for the new block, while the transactions of the new tree form the right subtree. This step is illustrated in Figure 2.4. This continuation anchors not only transactions inside a block into their order, but also chains blocks together with past blocks, giving the data structure the name *blockchain*.

2.4.5 Write optimized databases

By its very nature, the blockchain ledger is a append-only log. Furthermore, transactions typically modify the world state, pure queries, i.e. read-only accesses, are the exception. Therefore, blockchain systems tend to lean towards using write optimized databases for their internal data management. In particular, Hyperledger Fabric uses LevelDB, which we discuss in detail in this section.

Breaking down structured data storage to its most basic form, key-value stores consist of persistent dictionary data structures. Each entry has a unique key and a single blob value. The only restrictions usually posed to these keys and values are their byte lengths: in general, keys must be much shorter than values for increased lookup performance. Popular examples are Berkeley DB [86] and MemcacheDB [74]. Despite the relatively simplistic data structure, there is still ongoing development in this field and applications like Aerospike [109] have been optimized for current hardware trends like flash storage.

Those previously mentioned database systems are still geared towards a random-access-write architecture with in-place updates. However, when the data workload is highly skewed towards writes, they need to constantly jump between locations on the disk and therefore add a lot of overhead to the updates. To mitigate this, append-only logs guarantee high write throughput through sequential writes on disks where no lookups are necessary. But as the log grows, the reading speed of queried data diminishes drastically.

Databases based on log structured merge (LSM) trees [85] try to strike a balance where they optimize for write-heavy workloads without completely sacrificing read performance. LSM databases are layered into multiple levels, with a tree data structure in each of those levels. The unconventional idea behind these trees is that updated entries will not overwrite the old value in place, rather the new entry is simply appended and the old one "forgotten". A query then returns the first value it finds for a given key in the log in reverse write order, so older versions are obscured by newer ones. Every level of an LSM database has a specific size, typically the next higher level is at least twice as big. When a tree in a given level

reaches the current level's limit, its entries are merged into the next level. During these merges, obsolete versions of merged keys are garbage collected and the merged entries are appended to the tree in the next level. This procedure is shown schematically in Figure 2.5.

Google's BigTable [21] pioneered this kind of database. However, it opted to replace tree structures with Sorted String Tables (SSTables), files that consist of a list of key-values, sorted by keys. While this replaces tree sorting with list sorting, it does not change the functionality of the LSM database architecture. This in turn inspired LevelDB [29], its optimized Facebook fork RocksDB, HBase, Apache Cassandra and many others [95, 115, 27, 97]. LevelDB is used as the database back end of Hyperledger Fabric and is of specific interest for this proposal, therefore we will shortly discuss its structure here.

LevelDB collects new key-values in an unsorted hash table in main memory. When that table reaches a certain size it is transformed into an SSTable, but it is still held in memory in a level called L0. Level L0 can hold multiple SSTables with overlapping key ranges. When L0 reaches its maximum size or the maximum number of allowed SSTables, all L0 SSTables merge with the SSTables in the next level L1. During this merge, it is ensured that all newly created SSTables in L1 are sorted and have distinct key ranges. If in turn L1 now reaches its size threshold, L1 SSTables are merged with L2 SSTables in the same manner. This process continues iteratively until all levels reach equilibrium. Levels from L1 onward are stored on disk, although LevelDB can read chunks into memory and keep them in a buffer to improve read performance, if enough space is available.

2.5 Architecture

After the description of a blockchain system from a State Machine Replication point of view, we will now discuss more concretely the necessary steps nodes of a blockchain network have to go through to make progress. When a transaction is sent by a client of the system to a specific node, four steps must be executed to apply its changes to the world state.

Validation A transaction must be validated both syntactically as well as semantically. Syntactic validation ensures that a transaction is well formed, meaning it includes all necessary metadata and is transmitted in a known format. Semantic validation ensures that the state transition described by the transaction is valid within the constraints of the specific application, such as an account balance never dropping below zero. This also includes checking the validity of any signature and comparing hashes with the full data they represent to detect deviations.

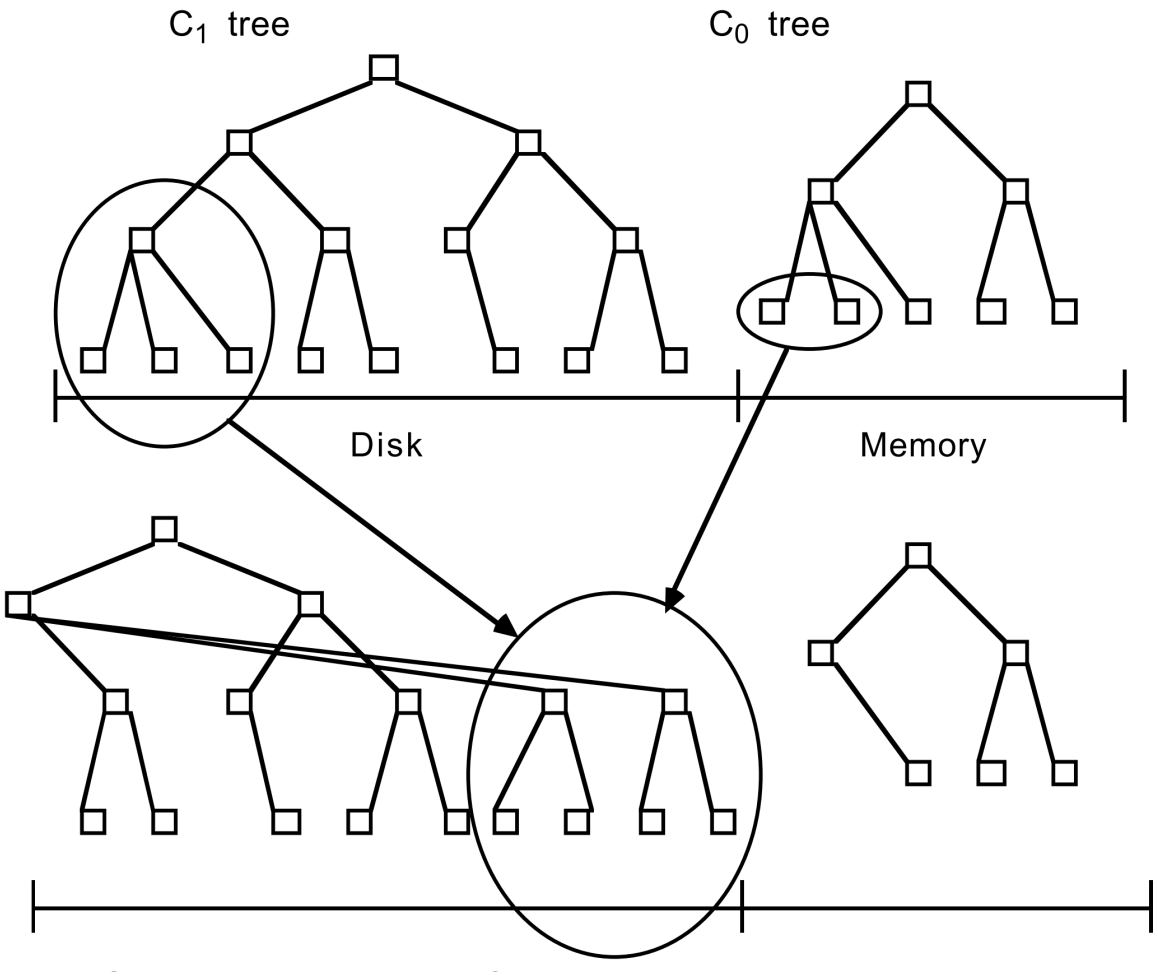


Figure 2.5: Conceptual picture of a rolling merge step of an LSM tree. The two highlighted nodes from the C_0 tree overlap with the four highlighted nodes of the C_1 tree. After the merge, all the highlighted nodes are discarded and replaced by a new sorted subtree that is appended to C_1 under the appropriate parent node [85].

Dissemination A (valid) transaction has to reach all nodes in the system. Depending on the consistency model of the specific blockchain implementation, there can be hard time constraints on the dissemination or the system can become eventually consistent, that is, consistent at some future unspecified point in time.

Consensus The nodes must agree that a transaction should be appended to the ledger. In an extension of the protocol, multiple transactions can be handled at once. In that case, consensus must also be reached on the order of those transactions.

Execution A valid transaction is applied to the current world state, whereby the specific state transition function depends on the application. Examples are a transfer of funds that results in the change of one or more account balances; the settlement of ownership of a real-world asset; or the execution of a contract. A detailed description of the data structures and management of the state and the ledger follows in Section 2.5.3.

All steps except consensus can be performed by nodes individually. The order of these steps can vary from implementation to implementation [3]. Especially the order of execution and consensus has far-reaching implications on the design of the blockchain, so we will discuss it further in the following.

2.5.1 The Order-Execute (OX) model

The *order-execute (OX)* approach guarantees consensus on the specific execution sequence of transactions in a block. However, it requires certain restrictions on the execution engine to guarantee that each node makes identical state transitions. First, the output of the execution engine must be deterministic. This requires the use of a deterministic contract language, such as Ethereum's Solidity [117], which must be learned by the application developer community. It also means that external oracles cannot easily be incorporated because different nodes in the network may receive different information from the oracle. Second, depending on the complexity of smart contracts, there needs to be a mechanism to deal with the *halting problem*, i.e., the inherent *a priori* unknowability of contract execution duration. A common solution to this problem is the inclusion of an execution fee like Ethereum's *gas*, which aborts long-running contracts.

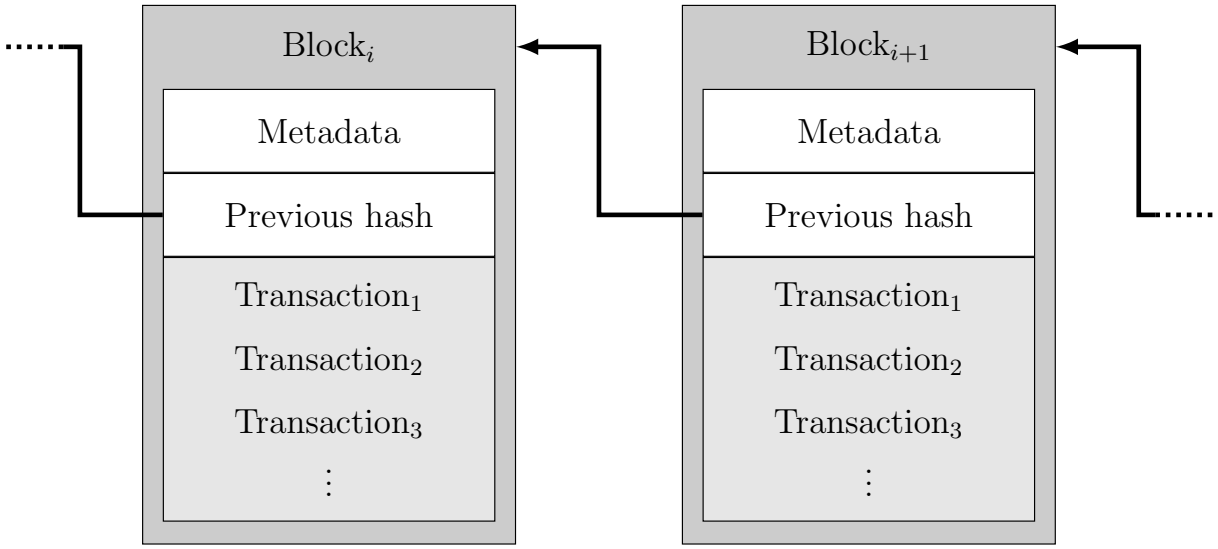


Figure 2.6: Simplified illustration of a blockchain.

2.5.2 The Execute-Order (XO) model

The *execute-order (XO)* model approach allows transactions to be executed in arbitrary order: the resulting state transitions are then ordered and aggregated into blocks. This allows transactions to be executed in parallel, increasing throughput. However, the world state computed at the time of state transition commitment is known to execution engines only after some delay, and all transactions are inevitably executed on a stale view of the world state. This makes it possible for transactions to result in invalid state transitions even though they were executed successfully before ordering. It necessitates a validation step after ordering so transitions can be invalidated deterministically based on detected conflicts.

OX and XO are diametric opposites in terms of concurrency control: OX pessimistically executes all transactions in the ordered sequence even if they are completely independent, while XO optimistically executes all transactions in parallel and later weeds out invalid results.

2.5.3 Data structure design

Blockchains implicitly use an *event sourcing pattern*, a term coined by Fowler [38]. This pattern describes the decoupling of the data storage from the data model the application

logic would use. In many long-lived applications the data model needs to change over time to fit business needs. If only the state of a system is stored in the database, these data either need to be transformed into the new model or complex mappings must be added. The former is difficult to complete without interruptions of a live system and the latter leads to maintenance problems down the road. With the event sourcing pattern, data are stored as events that change the system from one state to another. This event log is embedded in the blockchain ledger. The log itself is formed by the ordered list of transactions. However, each block also has to store the hash of the previous block to chain them together. It can also store implementation relevant metadata, like number of transactions in the block or time of creation. Furthermore, permissioned blockchains can store the signatures of the ordering service that created the block for authentication and verification. The ledger data structure is shown in Figure 2.6. The validity of data stored at a specific node can easily be verified by comparing the current Merkle root at this node with the root at another node. Even small changes buried deep in the tree propagate up to the root and change it completely. Therefore, tampering can be detected with a single hash comparison.

In the case of corruption of the local world state or simply when joining a new network, a node can always recover the current world state by replaying the events in the event log from the beginning. Moreover, views can be created on top of the event log that fit the current data model. Then, whenever a model change is necessary, a new view is created and the log is played back to populate it. A similar approach is used in Apache Spark's *Resilient Distributed Datasets*, for example [121].

As will be discussed in the following sections, some implementations even create artifacts corresponding to views in the event sourcing pattern, while others replay the log *ad hoc*.

While the definition of a transaction as a state transition message has been useful so far, when thinking about the data modeling and management it is better, perhaps, to focus on the types of data that are tracked on a blockchain and view transactions as the byproduct of their changes. There are four different types of data:

Asset metadata This type of data corresponds the closest with data stored in classic relational databases and is descriptive in nature. Examples are personal details like name, date of birth and address. A transaction might change these values, but then they are completely overwritten and only impact a single tracked entity. Usually, changes to metadata do not need to be validated against the versioning of the data and the last write wins.

Fungible assets This category generally represents a balance of some sort. An account could keep track of money or units of energy produced by a solar panel. These units are interchangeable and can be accumulated. In that case, a transaction describes a value difference instead of an absolute value. This difference, or *delta*, is then added or subtracted from the current account value at the time the state transition is being executed. Most importantly, for this type of data in addition to creation and destruction events, transactions can also be used to transfer a specific amount from one owner to another, affecting multiple entities simultaneously.

Before a transaction of fungible assets can be executed, it must be validated. Depending on the application, constraints need to be adhered to. For example, funds cannot be transferred if the funds in the buyer's account are less than the necessary amount for the transfer. Additionally, great care has to be taken in ensuring that the same funds will not be spent twice, a so-called double-spending attack. For both of these requirements, a state view reduces the amount computation for on-the-fly state reconstruction from the ledger.

Non-fungible assets This category represents unique goods. After creation, their intrinsic properties are fixed and they are clearly distinguishable from one another. Here, only extrinsic properties like ownership and location can change. Traversing the transaction log then becomes a trivial way of tracking provenance of a non-fungible asset. As with fungible assets, double spending attacks must be prevented, which is why a fast lookup of current ownership is crucial.

Smart contracts Even though they form the state transition function, their instantiation is tracked on the blockchain ledger as well, just like transactions. Otherwise, the availability of a smart contract in the network would not be deterministic. In that case, some nodes could view a transaction that calls a specific smart contract as valid and others as invalid, leading to an inconsistency in the network. A simple example would be to send a transaction with some amount of money and a list of beneficiaries as parameters to a smart contract. The smart contract then splits the amount into equal parts and sends a transaction to each of those beneficiaries, transferring their share.

2.6 Bitcoin

Famously envisioned by Satoshi Nakamoto [79], Bitcoin was the first blockchain. Its sole purpose is to track its namesake virtual currency on a permissionless chain. While block

miners are awarded some amount of Bitcoin when their proposed block becomes part of a valid chain, currency is kept scarce by an ever-diminishing reward function. Mining is achieved through a Proof of Work, specifically by solving a hash riddle, whose difficulty changes dynamically so that on average the global mining community proposes a block every ten minutes. This leads to an arms race between the algorithm and the miners. Nowadays, the most invested miners use powerful server farms with specific ASIC boards, thereby effectively negating any chance of mining success for the majority of the community. In 2018, Bitcoin mining surpassed the energy consumption of many smaller countries like Ireland or Denmark [73]. According to an estimation by the University of Cambridge [114], Bitcoin's energy consumption surpassed that of Switzerland in 2019 with over 60 TWh per year and is heading to consume more than 100 TWh in 2020.

While Bitcoin are essentially fungible, the blockchain does not keep track of the state of accumulated accounts. Transactions must reference previous transactions on the chain as proof for sufficient funds. On top of that, because of the missing view of the current system state, the validation of a new transaction would take too long if partial funds from past transactions could be used. Therefore, whenever a past transaction is referenced, its value must be consumed in its entirety. To that end, Bitcoin allows multiple recipients of a single transaction. So if one client wants to transfer an amount of Bitcoin to a different client by referencing a past transaction with a larger amount, the first client can add their own account as a secondary recipient, channeling back the remaining funds.

With its inherently small throughput, large latency due to its PoW consensus and without some kind of governance in place, Bitcoin is not interesting from a data management optimization point of view.

2.7 Ethereum

First described in its yellow paper by Wood [117], Ethereum has quickly become the second most popular permissionless blockchain for cryptocurrency. Ethereum still relies on a PoW consensus like Bitcoin. However, it has the advantage of an ardent core community with sway over the rest of the participants. In this way, even without built-in governance structures it is possible to update the network code; Ethereum 2.0 is currently under development and is slated to introduce a PoS solution named Casper [35]. In addition to its fungible currency Ether, the Ethereum chain allows the use of smart contracts, adding to its popularity. These contracts are written in the domain-specific and non-Turing-complete *Solidity* programming language to prevent security risks and allow reasoning about execution safety. The Ethereum specification explicitly demands the maintenance

of a system state view consisting of all accounts, called wallets, and their Ether and/or smart contract content. This is enforced by making the Merkle root of the current state part of the block proposal. While the specification leaves the details of data storage open, the most popular implementation, Go Ethereum or *Geth* for short [37], uses LevelDB as its database backend.

At the moment, Ethereum does not achieve a much higher throughput than Bitcoin, but it has a better chance to evolve and the possibility of smart contracts makes it the de facto leader for permissionless blockchains in any context that surpasses simple monetary exchange. It is expected that Ethereum will surpass Bitcoin as the number one cryptocurrency in the near future in an event dubbed *the flippening* [75].

2.8 Hyperledger Fabric

Founded by the Linux Foundation, Hyperledger is an umbrella project for several open source blockchains. Of all sub-projects, IBM's *Fabric* is the most actively developed and advanced incarnation. It is a permissioned blockchain that was built from the ground up with modularity in mind. We base the following description on the publication by Androulaki et al. [3], Fabric's documentation [49] and the source code [53]. Fabric provides a framework on top of which any kind of (permissioned) blockchain application can be created. Because of its permissioned nature, all nodes in the network must be known and registered with a membership service provider (MSP). Fabric uses X.509 certificates to identify nodes. Moreover, every node is assigned a specific role. By default these roles are *READER*, *WRITER* and *ADMIN*. Readers can only submit queries, writers are allowed to participate in the creation of new blocks and admins can additionally modify the network with tasks like adding nodes or updating the running code.

2.8.1 Node types

In Fabric, network nodes are either *orderers* or *peers*. The collection of orderers forms the ordering service that batches incoming transactions into blocks and sends them to the peers. Orderers are responsible solely for deciding transaction order, not correctness or validity. All peers commit blocks to a local copy of the blockchain and apply the corresponding changes to a *state database* that maintains a local snapshot of the current *world state*.

Each peer belongs to a specific *organization*. Generally, organizations represent the companies that collaborate on a given Fabric blockchain. Each organization must define

at least one *anchor peer*, which are known to the whole network. Once they are authenticated by the membership provider, other nodes in an organization can join the network by notifying their anchor peer. They are not necessarily known to peers outside of their organization. Besides defining peer visibility, organizations are also used as trust boundaries. Only anchor peers receive blocks from the ordering service, from where they are disseminated to the rest of the organization. This means, each peer implicitly trusts its anchor peer to be well-behaved. However, since the ordering service signs outgoing blocks, anchor peers can only withhold new blocks, they cannot falsify them.

Some peers are also *endorsers*. Endorser peers run instances of installed *chaincodes*, Fabric's version of smart contracts. This means only they can be called by *clients*, who create proposals for new transactions. Such proposals include an identifier for the target chaincode and any necessary calling parameter.

2.8.2 Artifacts

Fabric can reuse the same set of orderers and peers to support multiple blockchains. For each instance, Fabric creates a *channel*, which isolates its data from any other channel. Even though channels can run on the same hardware, they are completely independent of each other and there is no communication between them. Therefore, all our following descriptions of Fabric, if not explicitly stated otherwise, revolve around a single channel instance.

A channel's blockchain is split into two data structures. The transaction ledger itself is stored in data chunks in the file system. Fabric builds several indices on top of it to be able to find specific transactions in specific blocks quickly. These indices, as well as the world state, are stored in a state database. Implemented options for this task are CouchDB and LevelDB. In this thesis we focus on LevelDB, because it has a much higher performance [57]. LevelDB is a key-value store. Keys in this database follow the structure {`chaincode namespace`}{`separator symbol`}{`key name`}. The corresponding values are stored together with version labels. Because the most current version of a value is uniquely defined by the transaction which modified the respective key last, version labels are constructed by combining the block number in which the transaction occurred and its sequence number in the block's transaction order.

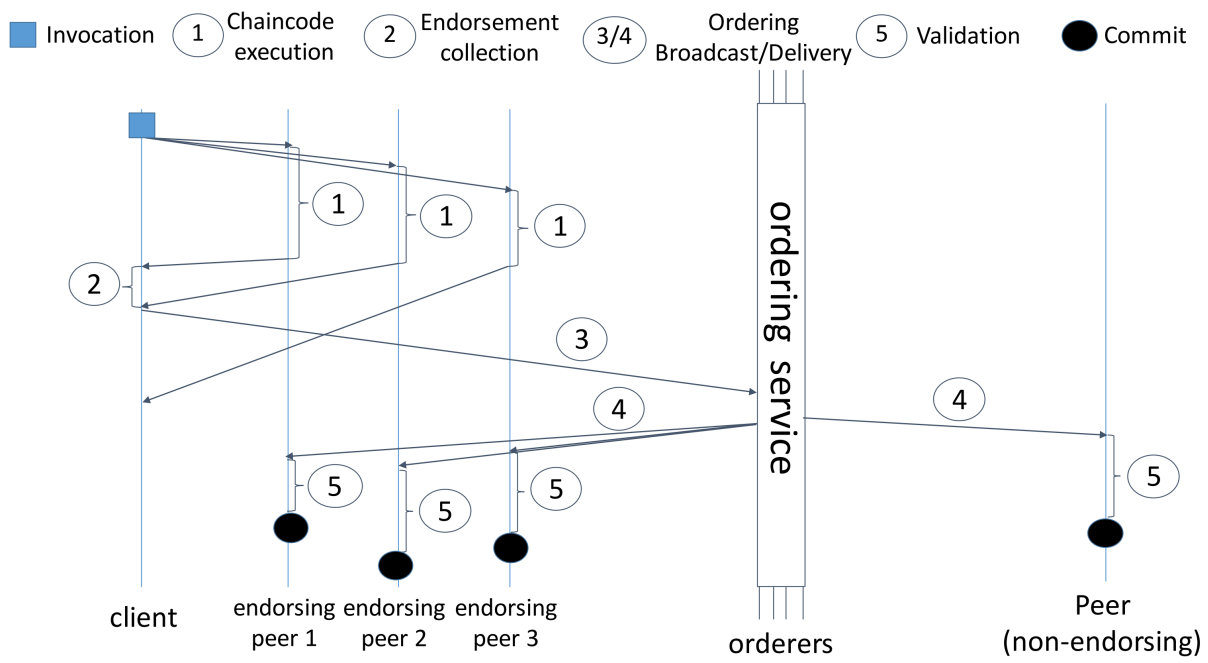


Figure 2.7: Illustration of the Hyperledger Fabric transaction flow [3]

2.8.3 Transaction flow

We now discuss the general transaction flow from proposal creation to commitment as illustrated in Figure 2.7.

To begin with, a *chaincode* must be installed on *endorsing peer* nodes. This chaincode translates the input from clients into instructions how to modify the world state. Additionally, all *peer* nodes, of which the endorsers are a subset, store a local copy of the ledger and world state.

Other blockchains such as Ethereum run into problems with non-deterministic smart contracts, because the contracts must be executed on all network nodes independently and return the same result. Therefore, they have to create domain-specific languages to curb such non-deterministic behavior. Fabric solves this problem elegantly by moving the code execution to the first step of the block creation. Even though developers can take full advantage of general purpose languages, deterministic execution is still guaranteed, because *client* nodes have to send a new transaction to multiple endorsing peers. The exact number is governed by the endorsement policy. This policy states which and how many endorsers

need to respond to the transaction proposal. It can declare specific endorsers or follow a more generic scheme like an N-out-of-M pattern. All of the endorsers then execute the chaincode concurrently. During this execution, they simulate the outcome by interacting with the ledger in a snapshot sandbox. They note the keys in the state database that must be read and written to in addition to the new values. Additionally, the expected version of the value of the read keys is kept. These read and write sets are added to the transaction proposal, the message is signed by the correspondent endorsing peer and sent back to the client. To sign the proposal response Fabric uses the Eliptic Curve Digital Signature Algorithm (ECDSA), but it can process RSA and AES as well [53, 39]. The client then collects all endorsements and when it has received sufficiently many according to the endorsement policy, it sends the original transaction proposal, the chaincode output and the endorsements to the *ordering service*.

The ordering service is a collection of *ordering nodes* that act as a completely decoupled consensus. They check that the client has a role that allows it to submit transactions and discards the submission otherwise. If everything is in order, the transaction will get queued for the consensus on new blocks of batched transactions. Each new block adds the hash of the previous block to its header. Fabric supports the use of multiple variants of SHA-2 and SHA-3 to create hashes and uses SHA-256 by default. The ordering service is modularized, so that the consensus algorithm can be easily swapped out for a different one. Current implementations include a no-consensus solution (single node orderer), as well as a Kafka-based and Raft-based non-BFT protocol. A BFT consensus protocol has also been proposed but is not integrated yet [110].

The ordering service broadcasts a new block to a set of known *anchor* peers. From there blocks get disseminated to the whole network via a gossip protocol. Peers then validate the syntactic soundness of each transaction as well as the adherence to the endorsement policy. Furthermore, they check that the read and write sets have not become stale. While endorsing peers do not endorse invalid transactions, it could happen that two transactions in the same block try to write to the same key in the state database. Since all state values are versioned, it is easy to notice conflicts with the expected version number in the read or write set and then discard that change. Based on these steps transactions in the block are marked as valid or invalid. The block is stored to the file system together with the validation flags and the state transitions for valid transactions are committed to the state database. As a consequence of that execution model, all transactions, even invalid ones, are kept in the transaction log, while only valid transactions actually change the state in the state database. This way, Fabric does not need an additional round of consensus for discarded transactions, even though the ordering service knew nothing of the semantic content of the block.

Chapter 3

FastFabric: Scaling transaction throughput

Blockchain technologies are expected to make a significant impact on a variety of industries. However, one issue holding them back is their limited transaction throughput, especially compared to established solutions such as distributed database systems. In this chapter, we re-architect Hyperledger Fabric to increase transaction throughput from 3,000 to 20,000 transactions per second. We focus on performance bottlenecks beyond the consensus mechanism, and we propose architectural changes that reduce computation and I/O overhead during transaction ordering and validation to greatly improve throughput. Notably, our optimizations, called FastFabric, are fully plug-and-play and do not require any interface changes to Hyperledger Fabric.

3.1 Implementation details

To set the stage for a discussion of the improvements in Section 3.2, we now take a closer look at the orderer and peer architecture.

3.1.1 Orderer

After receiving responses from endorsing peers, a client creates a *transaction proposal* containing a header and a payload. The header includes the Transaction ID and Channel

ID[1]. The payload includes the read-write sets and the corresponding version numbers, and endorsing peers' signatures. The transaction proposal is signed using the client's credentials and sent to the ordering service.

The two goals of the ordering service are (a) to achieve consensus on the transaction order and (b) to deliver blocks containing ordered transactions to the committer peers. Fabric currently uses Apache Kafka [63], which is based on ZooKeeper [48], for achieving crash-fault-tolerant consensus.

When an orderer receives a transaction proposal, it checks if the client is authorized to submit the transaction. If so, the orderer publishes the transaction proposal to a Kafka cluster, where each Fabric channel is mapped to a Kafka topic to create a corresponding immutable serial order of transactions. Each orderer then assembles the transactions received from Kafka into blocks, based either on the maximum number of transactions allowed per block or a block timeout period. Blocks are signed using the orderer's credentials and delivered to peers using gRPC [25].

3.1.2 Peer

On receiving a message from the ordering service a peer first unmarshals the header and metadata of the block and checks its syntactic structure. It then verifies that the signatures of the orderers that created this block conform to the specified policy. A block that fails any of these tests is immediately discarded.

After this initial verification, the block is pushed into a queue, guaranteeing its addition to the blockchain. However, before that happens, blocks go sequentially through two validation steps and a final commit step.

During the first validation step, all transactions in the block are unpacked, their syntax is checked and their endorsements are validated. Transactions that fail this test are flagged as invalid, but are left in the block. At this point, only transactions that were created in good faith are still valid.

In the second validation step, the peer ensures that the interplay between valid transactions does not result in an invalid world state. Recall that every transaction carries a set of keys it needs to read from the world state database (its read set) and a set of keys and values it will write to the database (its write set), along with their version numbers recorded by the endorsers. During the second validation step, every key in a transaction's read and write sets must still have the same version number. A write to that key from

[1]Fabric is virtualized into multiple *channels*, identified by the channel ID.

any prior transaction updates the version number and invalidates the transaction. This prevents double-spending.

In the last step, the peer writes the block, which now includes validation flags for its transactions, to the file system. The keys and their values, i.e., the world state, are persisted in either LevelDB or CouchDB, depending on the configuration of the application. Moreover, indices to each block and its transactions are stored in LevelDB to speed up data access.

3.2 Design

This section presents our changes to the architecture and implementation of Fabric version 1.2. This version was released in July 2018, followed by the release of version 1.3 in September 2018 and 1.4 in January 2019. However, the changes introduced in the recent releases do not interfere with our proposal, so we foresee no difficulties in integrating our work with newer versions. Importantly, our improvements leave the interfaces and responsibilities of the individual modules intact, meaning that our changes are compatible with existing peer or ordering service implementations. Furthermore, our improvements are mutually orthogonal and hence can be implemented individually. For both orderers and peers, we describe our proposals in ascending order from smallest to largest performance impact compared to their respective changes to Fabric.

3.2.1 Preliminaries

Using a Byzantine-Fault-Tolerant (BFT) consensus algorithm is a critical performance bottleneck in HyperLedger [116]. This is because BFT consensus algorithms do not scale well with the number of participants. In our work, we chose to look beyond this obvious bottleneck for three reasons:

- Arguably, the use of BFT protocols in permissioned blockchains is not as important as in permissionless systems because all participants are known and incentivized to keep the system running in an honest manner.
- BFT consensus is being extensively studied [7] and we expect higher-throughput solutions to emerge in the next year or two.
- In practice, Fabric 1.2 does not use a BFT consensus protocol, but relies, instead, on Kafka for transaction ordering, as discussed earlier.

For these reasons, the goal of our work is not to improve orderer performance using better BFT consensus algorithms, but to mitigate new issues that arise when the consensus is no longer the bottleneck. We first present two improvement to the ordering service, then a series of improvements to peers.

3.2.2 Orderer improvement I: Separate transaction header from payload

In Fabric 1.2, orderers using Apache Kafka send the entire transaction to Kafka for ordering. Transactions can be several kilobytes in length, resulting in high communication overhead which impacts overall performance. However, obtaining consensus on the transaction order only requires transaction IDs, so we can obtain a significant improvement in orderer throughput by sending only transaction IDs to the Kafka cluster.

Specifically, on receiving a transaction from a client, our orderer extracts the transaction ID from the header and publishes this ID to the Kafka cluster. The corresponding payload is stored separately in a local data structure by the orderer and the transaction is reassembled when the ID is received back from Kafka. Subsequently, as in Fabric, the orderer segments sets of transactions into blocks and delivers them to peers. Notably, our approach works with any consensus implementation and does not require any modification to the existing ordering interface, allowing us to leverage existing Fabric clients and peer code.

3.2.3 Orderer improvement II: Message pipelining

In Fabric 1.2, the ordering service handles incoming transactions from any given client one by one. When a transaction arrives, its corresponding channel is identified, its validity checked against a set of rules and finally it is forwarded to the consensus system, e.g. Kafka; only then can the next transaction be processed. Instead, we implement a pipelined mechanism that can process multiple incoming transactions concurrently, even if they originated from the same client using the same gRPC connection. To do so, we maintain a pool of threads that process incoming requests in parallel, with one thread per incoming request. A thread calls the Kafka API to publish the transaction ID and sends a response to the client when successful. The remainder of the processing done by an orderer is identical to Fabric 1.2.

Figure 3.1 summarizes the new orderer design, including the separation of transaction IDs from payloads and the scale out due to parallel message processing.

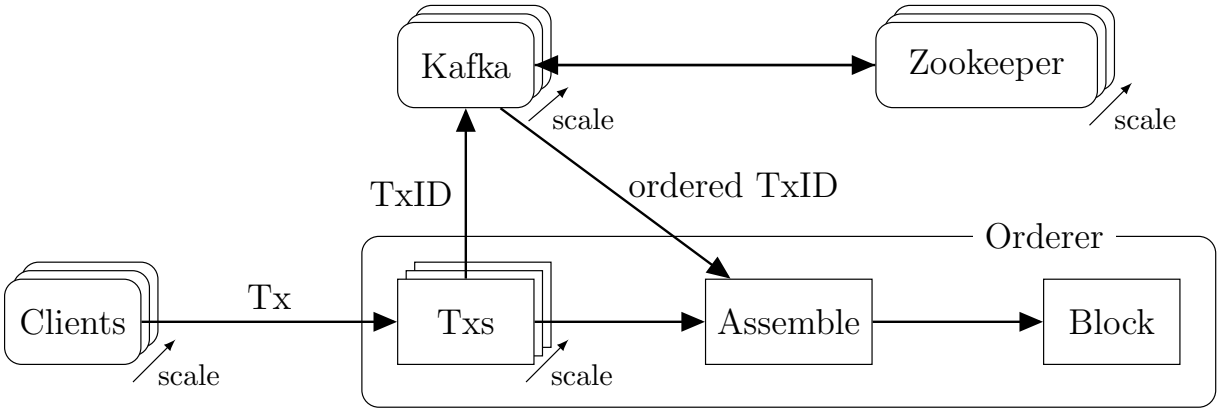


Figure 3.1: New orderer architecture. Incoming transactions are processed concurrently. Their TransactionID is sent to the Kafka cluster for ordering. When receiving ordered TransactionIDs back, the orderer reassembles them with their payload and collects them into blocks.

3.2.4 Peer tasks

Recall from Section 3.1.2 that on receiving a block from an endorser, a Fabric peer carries out the following tasks in sequence:

- Verify legitimacy of the received message
- Validate the block header and each endorsement signature for each transaction in the block
- Validate read and write sets of the transactions
- Update the world state in either LevelDB or CouchDB
- Store the blockchain log in the file system, with corresponding indices in LevelDB

Our goal is to maximize transaction throughput on the critical path of the transaction flow. To this end, we performed extensive tests described in detail in Section 3.4 to identify performance bottlenecks.

We make the following observations. First, the validation of a transaction's read and write set needs fast access to the world state. Thus, we can speed up the process by using an in-memory hash table instead of a database (Section 3.2.5). Second, the blockchain log

is not needed for the transaction flow, so we can defer storing it to a dedicated storage and data analytics server at the end of the transaction flow (Section 3.2.6). Third, a peer needs to process new transaction proposals if it is also an endorser. However, the committer and endorser roles are distinct, making it possible to dedicate different physical hardware to each task (Section 3.2.7). Fourth, incoming blocks and transactions must be validated and resolved at the peer. Most importantly, syntactic and cryptographic transaction validations are completely independent of each other. This allows the creation of a fully parallelized validation pipeline (Section 3.2.8).

Finally, significant performance gains can be obtained by caching the results of the *Protocol Buffers* [40] unmarshaling of blocks (Section 3.2.9). We detail this architectural redesign, including the other proposed peer improvements, in Figure 3.2.

3.2.5 Peer improvement I: Replacing the world state database with a hash table

The world state database must be looked up and updated *sequentially* for each transaction to guarantee consistency across all peers. Thus, it is critical that updates to this data store happen at the highest possible transaction rate.

We believe that for common scenarios, such as for tracking of wallets or assets on the ledger, the world state is likely to be relatively small. Even if billions of keys need to be stored, most servers can easily keep them in memory. Therefore, we propose using an in-memory hash table, instead of LevelDB/CouchDB, to store world state. This eliminates hard drive access when updating the world state. It also eliminates costly database system guarantees (i.e., ACID properties) that are unnecessary due to redundancy guarantees of the blockchain itself, further boosting the performance. Naturally, such a replacement is susceptible to node failures due to the use of volatile memory, so the in-memory hash table must be augmented by stable storage. We address this issue in Section 3.2.6.

3.2.6 Peer improvement II: Store blocks using a peer cluster

A regular peer in the Fabric network has two distinct tasks, the computation-heavy transaction validation and the IO-heavy data storage. Importantly, validation and any interaction with the world state do not rely directly on the data stores on the hard drive. With the state database cached in a hash table as described in Section 3.2.5 we are now free to move the persistent storage of both the blockchain log and the state database to different hardware.

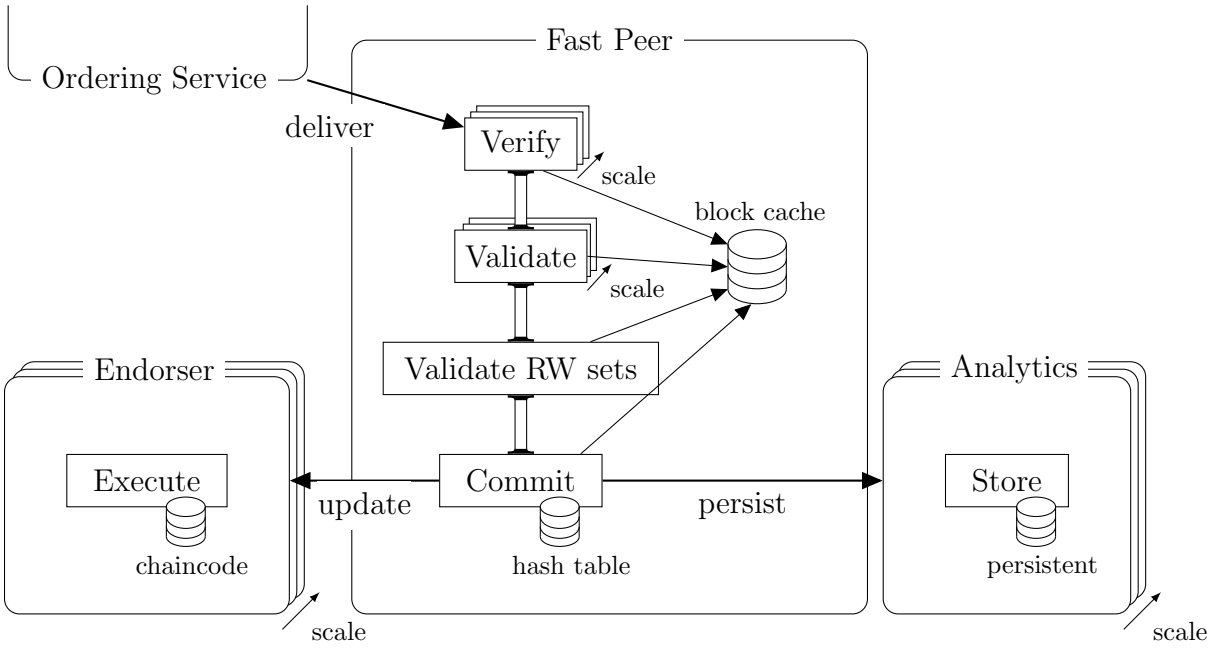


Figure 3.2: New peer architecture. The fast peer uses an in-memory hash table to store the world state. The validation pipeline is completely concurrent, validating multiple blocks and their transactions in parallel. The endorser role and the persistent storage are separated into scalable clusters and given validated blocks by the fast peer. All parts of the pipeline make use of unmarshaled blocks in a cache.

In our proof of concept implementation we move LevelDB for the world state and the file system storage for the blockchain log to a different server. After this change, the peer server validates blocks and transactions, updates its world state cache and then sends the validated blocks to the storage server. The peer can use this persistent storage server to recover its world state cache after a crash. However, such a mechanism is beyond the scope of this work since we focus on improving Fabric's maximum performance and not recovery from crash failures.

After decoupling computation and storage we can envision many types of data stores for blocks and world state backups, including a distributed storage cluster. In such a cluster, each storage server would only contain a fraction of the chain, motivating the use of distributed data processing tools such as Hadoop MapReduce or Spark.

Note that the storage system is only visible to the peer server. For the rest of the network the peer still appears as a single entity.

3.2.7 Peer improvement III: Separate commitment and endorsement

In Fabric 1.2, every endorser peer is simultaneously a committing peer. Endorsement is an expensive operation, as is commitment. When fully utilizing existing hardware resources, endorsers get into a paradoxical situation. On the one hand an additional endorser in the network allows for more transactions to be executed in parallel, meaning the initial transaction throughput can be increased. On the other hand, increased transaction throughput means the commitment step needs more resources and thereby throttles the endorsement which is running on the same server. Because of this competition for resources the net gain for every new endorser peer in the network shrinks to the point where the overall throughput decreases rather than increases. Therefore, we propose to split these roles to separate hardware.

Specifically, in our design, a committer peer processes the validation pipeline and then sends validated blocks to a cluster of endorsers that only apply the changes to their world state without further validation. This step allows us to free up resources on the peer.

As with the storage cluster, this new endorser cluster should appear to still be a part of the peer to the rest of the network. This means, the peer server must incorporate a load balancer to distribute transactions among the endorsers. Because none of the endorser servers has to do block validation, the peer can scale out its endorser cluster to meet demand.

In our proof of concept we omitted the automatic load balancing step and matched clients manually to different endorser servers.

3.2.8 Peer improvement IV: Parallelize validation

Both block and transaction header validation, which include checking permissions of the sender, enforcing endorsement policies and syntactic verification, are highly parallelizable. We extend the concurrency efforts of Fabric 1.2 by introducing a complete validation pipeline.

Specifically, for each incoming block, one go-routine is allocated to shepherd it through the block validation phase. Subsequently, each of these go-routines makes use of the go-routine pool that already exists in Fabric 1.2 for transaction validation. Therefore, at any given time, multiple blocks and their transactions are checked for validity in parallel. Finally, all read-write sets are validated sequentially by a single go-routine in the correct order. This enables us to utilize the full potential of multi-core server CPUs.

3.2.9 Peer improvement V: Cache unmarshaled blocks

Fabric uses gRPC for communication between nodes in the network. To prepare data for transmission, *Protocol Buffers* are used for serialization. To be able to deal with application and software upgrades over time, Fabric's block structure is highly layered, where each layer is marshaled and unmarshaled separately. This leads to a vast amount of memory allocated to convert byte arrays into data structures. Moreover, Fabric 1.2 does not store previously unmarshaled data in a cache, so this work has to be redone whenever the data is needed.

To mitigate this problem, we propose a temporary cache of unmarshaled data. Blocks are stored in the cache while in the validation pipeline and retrieved by block number whenever needed. Once any part of the block becomes unmarshaled, it is stored with the block for reuse. We implement this as a cyclic buffer that is as large as the validation pipeline. Whenever a block is committed, a new block can be admitted to the pipeline and automatically overwrites the existing cache location of the committed block. As the cache is not needed after commitment and it is guaranteed that a new block only arrives after an old block leaves the pipeline, this is a safe operation. Note that unmarshaling only adds data to the cache, it never mutates it. Therefore, lock-free access can be given to all go-routines in the validation pipeline. In a worst-case scenario, multiple go-routines try to access the same (but not yet unmarshaled) data and all proceed to execute the unmarshaling in parallel. Then the last write to the cache wins, which is not a problem because the result would be the same in either case.

3.3 FastFabric failure model

With the full architecture of FastFabric in place, we have introduced additional hardware and software processes, which changes Fabric's original failure model. FastFabric introduces a *peer cluster* in place of Fabric's single peer node, as shown in Figure 3.2. A FastFabric peer cluster consists of a single fast peer, an endorser cluster and a storage cluster each consisting of one or more servers. Note that each node in a Fabric network would be replaced by an independent peer cluster.

We start with a discussion of stop-failures. Because the peer cluster logically still represents a single peer, its hardware should be localized, possibly within a single rack in a data center. This means we can rely on best practices of data center maintenance such as hardware redundancy and virtualization to minimize the risk of hardware failures [8, 72, 56]. Next, we focus on the consequences of any member of the cluster failing and consider endorsers, the fast peer and the storage cluster in turn. Because endorsers are

exact replicas of each other, FastFabric is more robust to endorser failures than default Fabric. Indeed, as long as a single endorser server is alive, a peer cluster's functionality is unimpeded, only its endorsement throughput degrades.

FastFabric's fast peer shares Fabric's single peer failure model. That is, if a fast peer encounters a failure, the whole peer cluster fails because the other parts rely on the fast peer for data replication. However, this only means that a single peer (cluster) in the network becomes unavailable. The network should be set up with enough peer clusters so that data is stored redundantly and the chance of a catastrophic failure is negligible.

The failure model of the storage cluster depends on the chosen storage solution. For example, with a distributed database, the number of tolerable faults depends on its replication factor. If, instead, data is stored in shards without replication or a single server is used, then any failure leads to data loss. In this case, it might be possible to recover the data from the associated peer cluster's fast peer, if it is still in the fast peer's cache. Otherwise, the data needs to be recovered from other peer clusters in the network. Until the recovery is completed, the whole peer cluster must ignore client queries and new blocks to prevent data inconsistencies. Given that the failure of a storage server without replication stops the entire peer cluster until the data loss is repaired, it should be considered best practice to configure peer clusters to use storage clusters with replication to counteract the increased possibility of hardware faults due to the larger number of servers in the system.

Now, consider external attacks on each portion of the peer cluster in turn as the cause of failures. To begin, note that the storage cluster only serves as persistent data lookup for the fast peer and endorser cluster, so there is no need to make them publicly available. This leaves only the possibility of an attack through physical access. In this case, the attacker most probably has equal opportunity to attack any of the peer cluster's components. Due to the peer cluster's localized server installation, gaining physical access to one server is equivalent to gaining access to all servers in the peer cluster. Consequently, the probability of attacking a FastFabric peer cluster in this manner is the same as attacking a single Fabric peer. If the fast peer becomes corrupted through an attack, then the whole peer cluster fails, but other peers in the network are unimpeded. Lastly, if one of the endorsers becomes the successful target of an attack, then it might start to create false responses to transaction proposals. This can be caught by a properly set up endorsement policy which requires multiple endorsers to agree on a result. However, since the endorsers of one peer cluster are exact replicas, we have to assume that if one endorser can be attacked then all of them can. Therefore, it is crucial that all endorsers sign responses with the same private key, i.e., use the same identity. Otherwise, multiple endorser servers of the same corrupted peer cluster could pose for different peer clusters, increasing the risk that the malicious computation satisfies the assigned endorsement policy.

To summarize, the failure model for a Fabric *peer* and a FastFabric *fast peer* are identical. Moreover, a properly set up FastFabric peer cluster that uses the same identity for all its parts and implements some kind of replication on its storage servers is more reliable regarding failures of its endorsers and storage servers than Fabric's single peer node.

3.4 Preliminary experiments

To ensure the maximum performance impact of any changes to Fabric's source code we carefully analyzed the bottlenecks of Fabric 1.2. In the following, we will describe the techniques we applied and the preliminary results that lead us to develop the optimizations described in Section 3.2.

For all described experiments, the network setup consists of a single endorsement peer and one orderer. The installed chaincode transfers digital coins from one account to another. Keys in the state database represent accounts that store their current balance and each transaction must read the value of two accounts and then write back the updated value after the transfer, so the read and write sets contain the same two keys. The endorsement policy requires a single endorsement from any endorsing peer and 100 transactions are ordered into each block. We use two servers equipped with Intel® Xeon® CPU E5-2620 v2 processors at 2.10 GHz, for a total of 24 hardware threads, 64 GB of RAM and an SSD hard drive.

3.4.1 Call graph analysis

As a first step to understand the performance of Fabric, we investigate the critical path on the peer in its completeness. The major elements of the path are:

- Block delivery from an orderer to the peer in the form of a Protobuffer byte stream via gRPC
- Unmarshaling of all layers of Protobuffers that comprise a block and its transactions
- Syntactic verification of each block
- Syntactic verification of each transaction
- Validation of the endorsement policy of each transaction

- Storage of the block in the file system
- Validation of read and write set conflicts for each transaction in a block
- Modification of the current world state stored in LevelDB based on the write sets of all valid transactions

To get a sense of how much computing resources each of these steps takes up we use the Go language's integrated profiling tool *pprof* to create a CPU profile of the peer. The profiler interupts the peer process in short intervals to record a *sample* with information about the current stack trace. An accurate picture of where most CPU cycles are spent is formed over time by accumulating these samples. Because the optimization of chaincode execution and endorsement is beyond the scope of this work, we exclude all samples that were recorded in those contexts. The call graph in Figure 3.3 graphically represents the result of the profiling.

We exclude all samples not directly related to the critical path execution. The remaining ones make up 21.68% of the total samples[2]. This should not be taken as a sign that the critical path uses a minor amount of resources. Rather, it takes time to set up the experiments and many samples are taken in an idle state before the start of the tests. Additionally, the chaincode execution takes up a significant amount of resources. However, we are only interested in the critical path for validation and commitment. Therefore, we normalize the critical path and show the scaled up CPU load percentages of the major contributors in Table 3.1. The four biggest distinct system elements make up 75.67% of the overall execution time, guiding the focus of our efforts.

Note that this load distribution is specific to the setup of the experiment. For example, a more elaborate endorsement policy would shift the distribution to a heavier emphasis on

[2]We get this by summing the cumulative percentages of the two source nodes *Validate* and *StoreBlock* in the call graph

Table 3.1: Percentages of the overall execution time of the critical path on the peer in the default Fabric 1.2 implementation.

Call graph nodes	Critical path samples (%)
LevelDB	30.21
Protobuffer unmarshaling	29.28
Cryptographic operations	10.93
Memory allocation	5.25
	75.67

the cryptographic operations. The described setup was chosen to put a relatively balanced load on all parts of the peer's processing pipeline to highlight multiple starting points for optimizations.

We address the high impact of LevelDB (Section 3.2.5) and Protobuffer unmarshaling (Section 3.2.9) directly. However, substituting all cryptographic implementations with more efficient ones is a highly complex undertaking and instead, we opted to circumvent this bottleneck by maximizing the parallel execution of the cryptographic validation (Section 3.2.8).

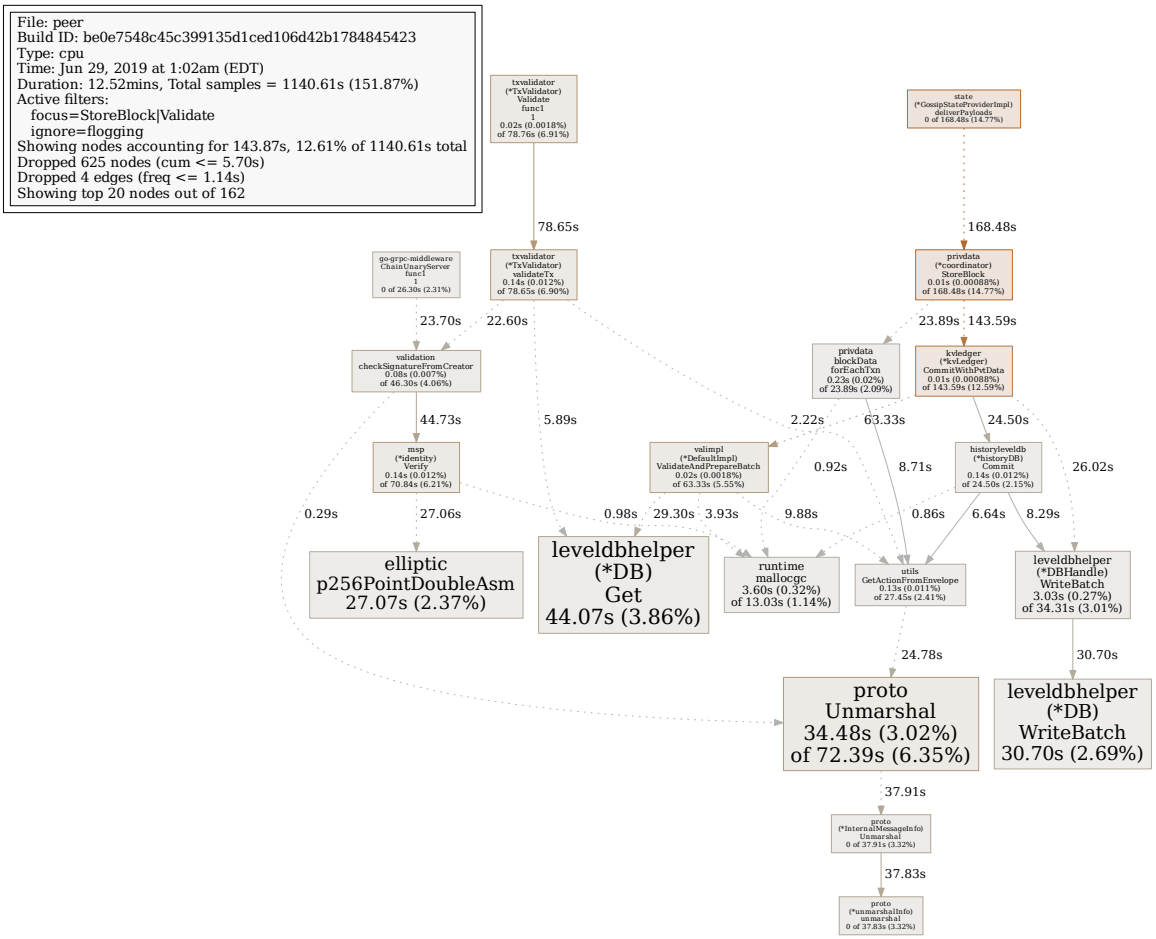


Figure 3.3: Call graph of the peer of default Fabric 1.2. Only top 20 nodes concerning commitment and validation are shown.

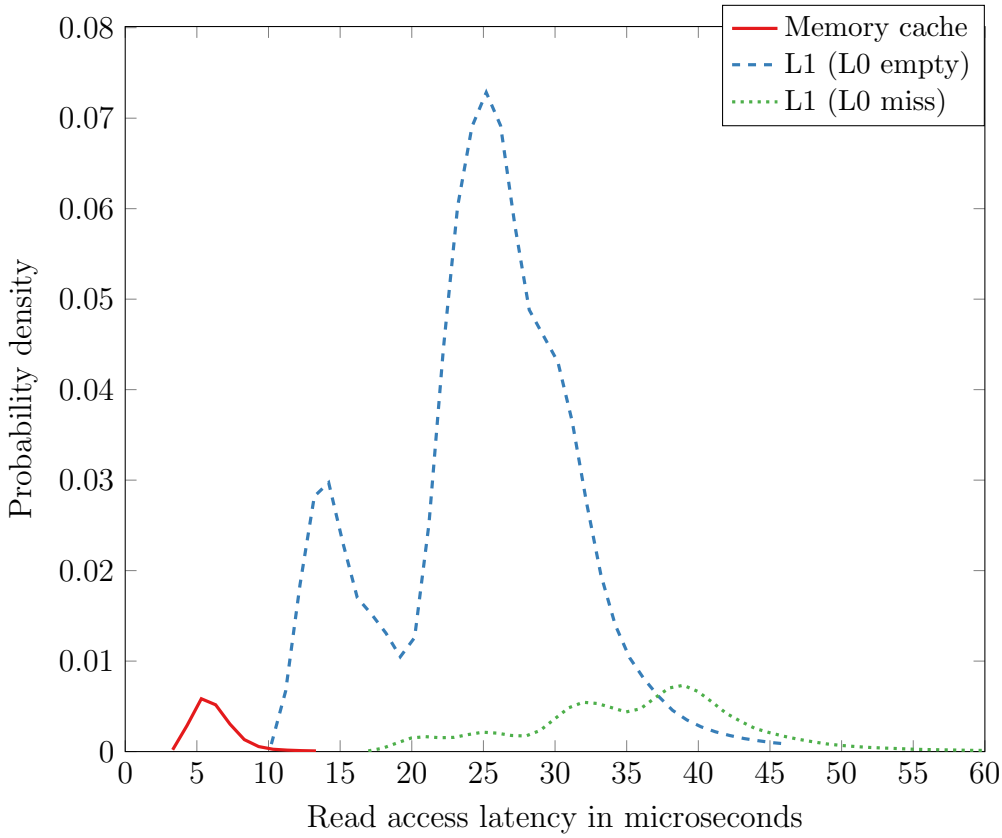


Figure 3.4: Benchmark of LevelDB read latencies for a memory cache size of 1MB. It shows the distribution after 100 million transactions have made updates to 10 million keys. The majority of accesses hit the memory buffer of L1 while L0 is completely empty.

3.4.2 LevelDB

Accessing the state database stored in LevelDB takes up almost 40% of the critical path's execution time, so we prioritized the search for a solution for this bottleneck. As a first approach, we explored an optimization by parameter tuning.

Because every key in a transaction's read and write set has to be validated against the state database, all of these keys must be read from LevelDB. Therefore, we look at read latencies as a metric for performance. For our benchmarks, we initiate LevelDB with 10 million keys, send 100 million transactions which make use of these account keys through Fabric's read and write set validation and commitment step and record key read latencies. Exploring the impact of multiple tuning parameters like memory cache size,

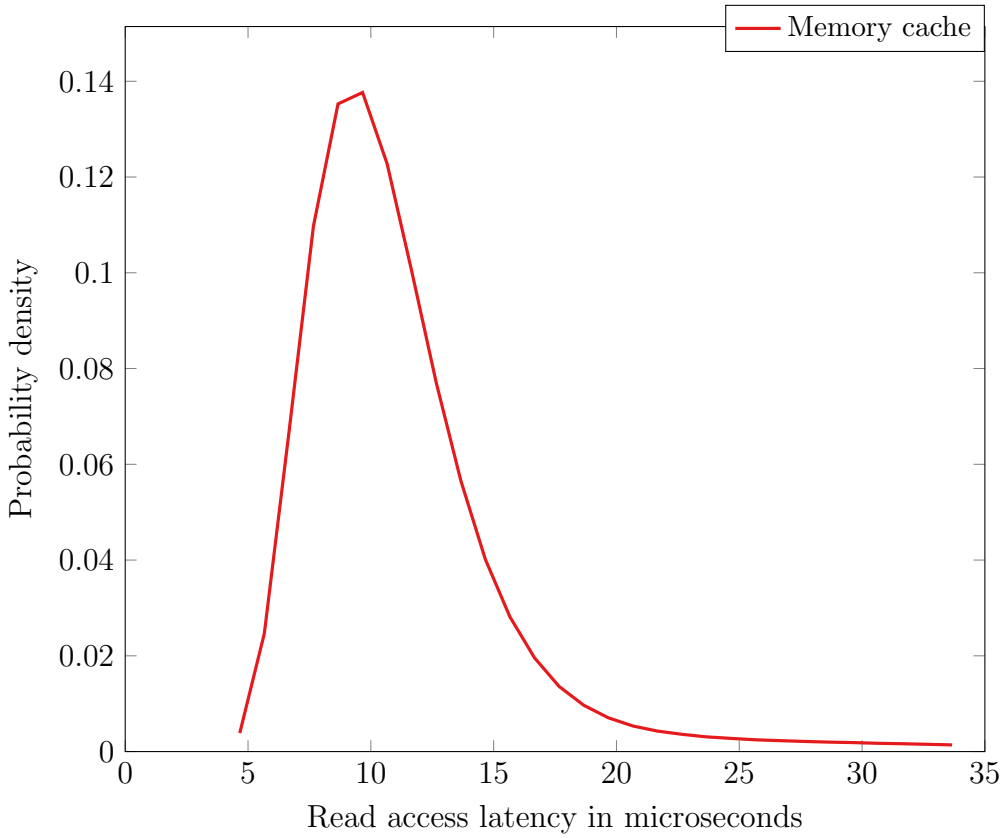


Figure 3.5: Benchmark of LevelDB read latencies for a memory cache size of 256MB. It shows the distribution after 100 million transactions have made updates to 10 million keys. All keys fit into the memory cache, so no other levels are hit.

number of tables per level and transaction throughput, we find the only parameter to make a noticeable difference on LevelDB's performance is the cache size.

We present the results for the smallest and largest cache sizes we tested and compare them. Figure 3.4 shows the distribution for a cache size of 1MB, the smallest configuration we tested. We make several observations: The whole key space fits into levels up to L1 and no data was ever written to L2. Additionally, all data was buffered into main memory after a short warm-up period. Whenever a compaction to L1 happened, the buffered data was updated simultaneously to the update to the hard drive. This means, all distributions shown in Figure 3.4 are main memory accesses. Furthermore, whenever LevelDB is idle, L0 is compacted into L1. Because of this, L0 was empty most of the time during our

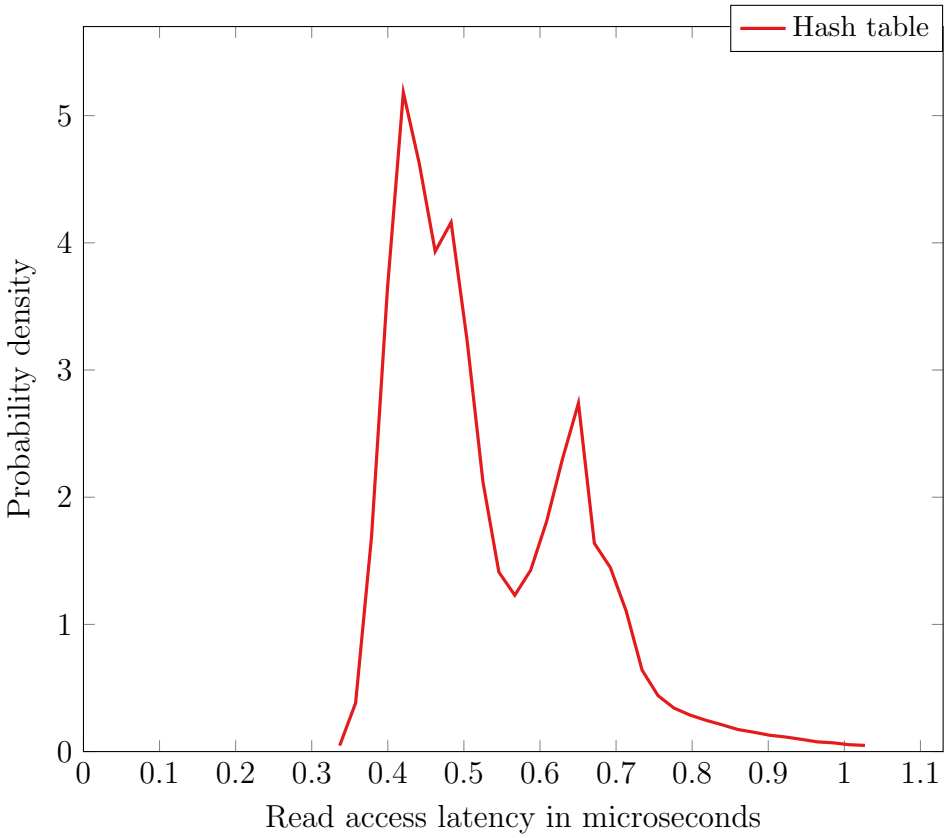


Figure 3.6: Read performance benchmark of 100 million transactions against a light-weight hash table with LevelDB API and readers-writer lock for concurrent access, implemented in GO. Results outperform LevelDB by almost two orders of magnitude.

benchmarks. This leaves three scenarios to make up the bulk of all accesses. In the first case, the key was found in the memory cache (red line). In the second case, it was neither found in the cache nor in L0 (black dotted line). In the third case, it was not found in the cache and L0 was empty (blue dashed line). The third scenario is faster than the second one because L0 did not need to be scanned. The average read access to a key in this configuration is about 30 microseconds.

The larger the size of the memory cache, the more frequently a key is found there. At the size of 256MB, the entire key space fits into the memory cache, which is shown in Figure 3.5. This decreases the read access latency to about 12 microseconds because none of the buffers for the higher levels need to be accessed.

To measure how much computational overhead is involved in LevelDB's read accesses, we implemented a lightweight hash table in Go with the same API as LevelDB and a simple readers-writer lock for concurrent access. Then, we used it as a substitute for LevelDB as the Fabric back end and repeated the benchmark. This setup consistently achieves nearly two orders of magnitude of performance improvement, as seen in Figure 3.6.

3.5 Results

This section presents an experimental performance evaluation of our architectural improvements. Our setup comprises fifteen local servers connected by a 1 Gbit/s switch. Each server is equipped with two Intel® Xeon® CPU E5-2620 v2 processors at 2.10 GHz, for a total of 24 hardware threads and 64 GB of RAM. We use Fabric 1.2 as the base case and add our improvements step by step for comparison. By default, Fabric is configured to use LevelDB as the peer state database and the orderer stores completed blocks in-memory, rather than on disk. Furthermore, we run the entire system without using docker containers to avoid additional overhead.

While we ensured that our implementation did not change the validation behaviour of Fabric, all tests were run with non-conflicting and valid transactions. This is because valid transactions must go through every validation check and their write sets will be applied to the state database during commitment. In contrast, invalid transactions can be dropped. Thus, our results evaluate the worst case performance.

For our experiments which focus specifically on either the orderer or the committer, we isolate the respective system part. In the orderer experiments, we send pre-loaded endorsed transactions from a client to the orderer and have a mock committer simply discard created blocks. Similarly, during the benchmarks for the committer, we send pre-loaded blocks to the committer and create mocks for endorsers and the block store which discard validated blocks.

Then, for the end-to-end setup, we implement the full system: Endorsers endorse transaction proposals from a client based on the replicated world state from validated blocks of the committer; the orderer creates blocks from endorsed transactions and sends them to the committer; the committer validates and commits changes to its in-memory world state and sends validated blocks to the endorsers and the block storage; the block storage uses the Fabric 1.2 data management to store blocks in its file system and the state in LevelDB. We do not, however, implement a distributed block store for scalable analytics; that is beyond the scope of this work.

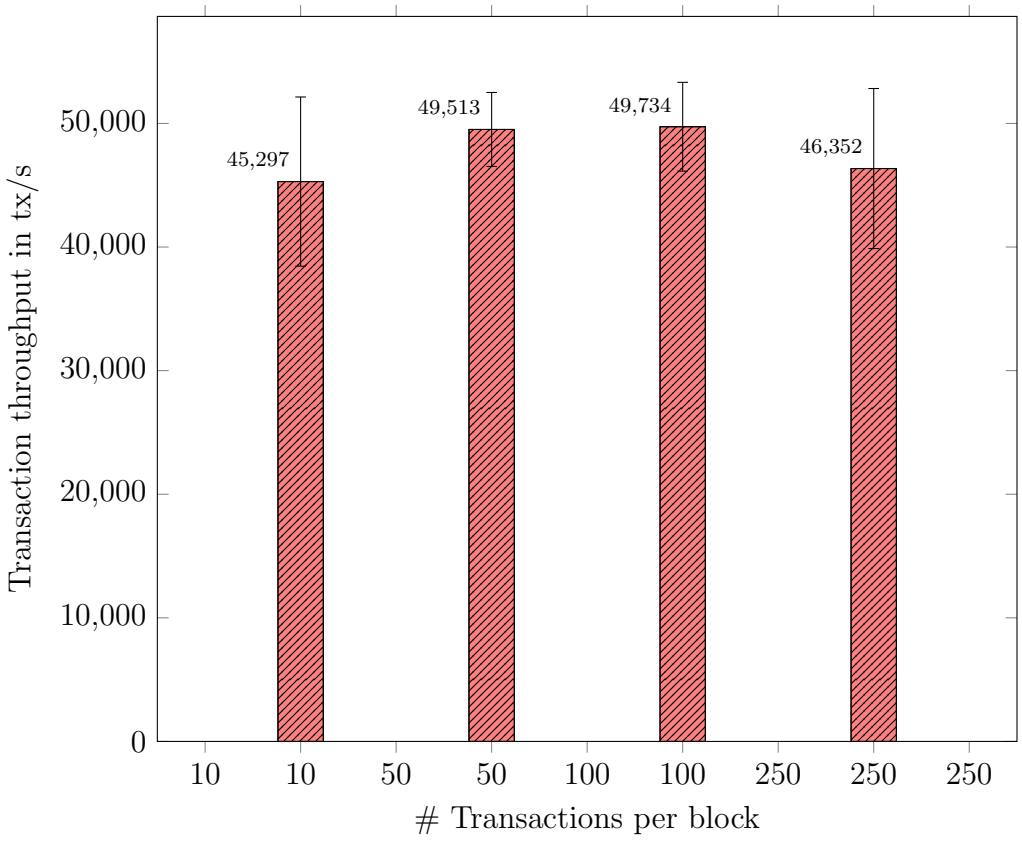


Figure 3.7: Throughput via gRPC for different block sizes.

For a fair comparison, we used the same transaction chaincode for all experiments: Each transaction simulates a money transfer from one account to another, reading and making changes to two keys in the state database. These transactions carry a payload of 2.9 KB, which is typical [108]. Furthermore, we use the default endorsement policy of accepting a single endorser signature.

3.5.1 Block transfer via gRPC

We start by benchmarking the gRPC performance. We pre-created valid blocks with different numbers of transactions in them, sent them through the Fabric gRPC interface from an orderer to a peer, and then immediately discarded them. The results of this experiment are shown in Figure 3.7.

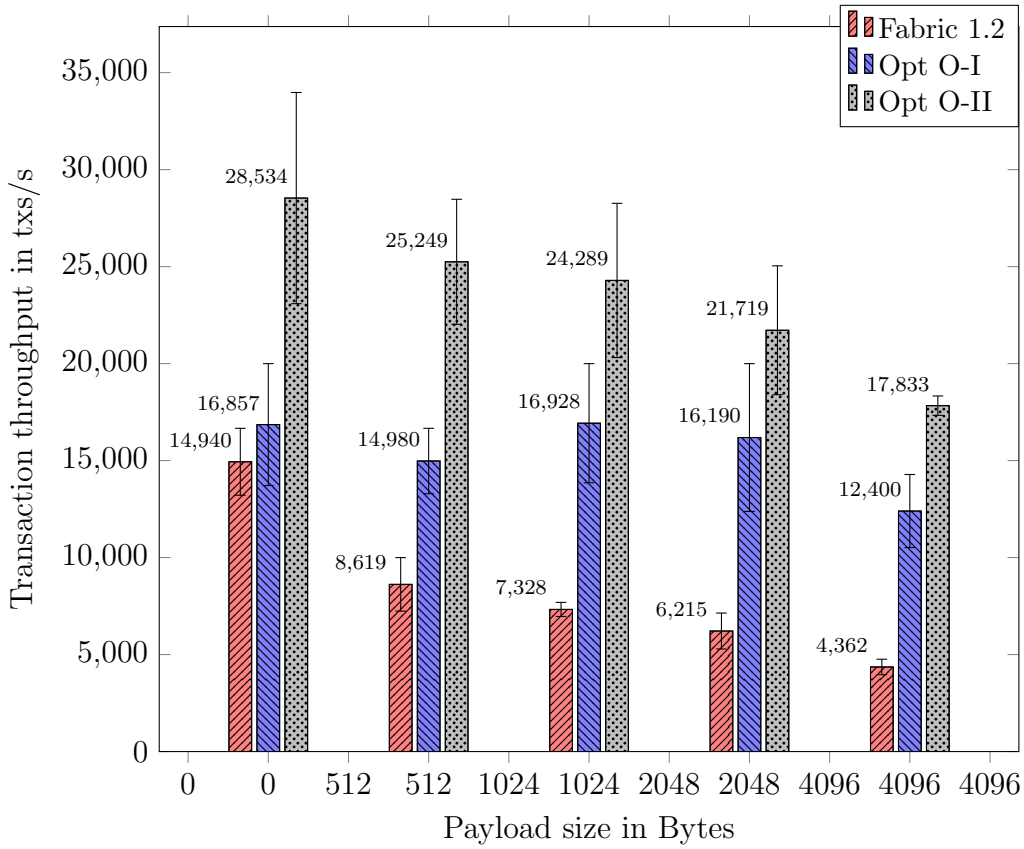


Figure 3.8: Effect of payload size on orderer throughput.

We find that for block sizes from 10 to 250 transactions, which are the sizes that lead to the best performance in the following sections, a transaction throughput rate of more that 40,000 transactions/s is sustainable. Comparing this with the results from our end-to-end tests in Section 3.5.4, it is clear that in our environment, network bandwidth and the 1 Gbit/s switch used in the server rack are not the performance bottlenecks.

3.5.2 Orderer throughput as a function of message size

In this experiment, we set up multiple clients that send transactions to the orderer and monitor the time it takes to send 100,000 transactions. We evaluate the rate at which an orderer can order transactions in Fabric 1.2 and compare it to our improvements:

- **Opt O-I**: only Transaction ID is published to Kafka (Section 3.2.2)
- **Opt O-II**: parallelized incoming transaction proposals from clients (Section 3.2.3)

Figure 3.8 shows the transaction throughput for different payload sizes. In Fabric 1.2, transaction throughput decreases as payload size increases due to the overhead of sending large messages to Kafka. However, when we send only the transaction ID to Kafka (Opt O-1), we can almost triple the average throughput (2.8×) for a payload size of 4096 KB. Adding optimization O-2 leads to an average throughput of 4× over the base Fabric 1.2. In particular, for the 2 KB payload size, we increase orderer performance from 6,215 transactions/s to 21,719 transactions/s, a ratio of nearly 3.5x.

3.5.3 Peer experiments

In this section, we describe tests on a single peer in isolation (we present the results of an end-to-end evaluation in Section 3.5.4). Here, we pre-computed blocks and sent them to a peer as we did in the gRPC experiments in Section 3.5.1. The peer then completely validates and commits the blocks.

The three configurations shown in the figures compared to Fabric 1.2 *cumulatively* incorporate our improvements (i.e., Opt P-II incorporate Opt P-I, and Opt P-III incorporates both prior improvements):

- **Opt P-I** LevelDB replaced by an in-memory hash table
- **Opt P-II** Validation and commitment completely parallelized; block storage and endorsement offloaded to a separate storage server via remote gRPC call
- **Opt P-III** All unmarshaled data cached and accessible to the entire validation/commitment pipeline.

Experiments with fixed block sizes

Figure 3.9 and 3.10 show the results from validation and commitment of 100,000 transactions for a single run, repeated 1,000 times. Transactions were collected into blocks of 100 transactions[3]. We first discuss latency, then throughput.

[3]We experimentally determined that peer throughput was maximized at this block size.

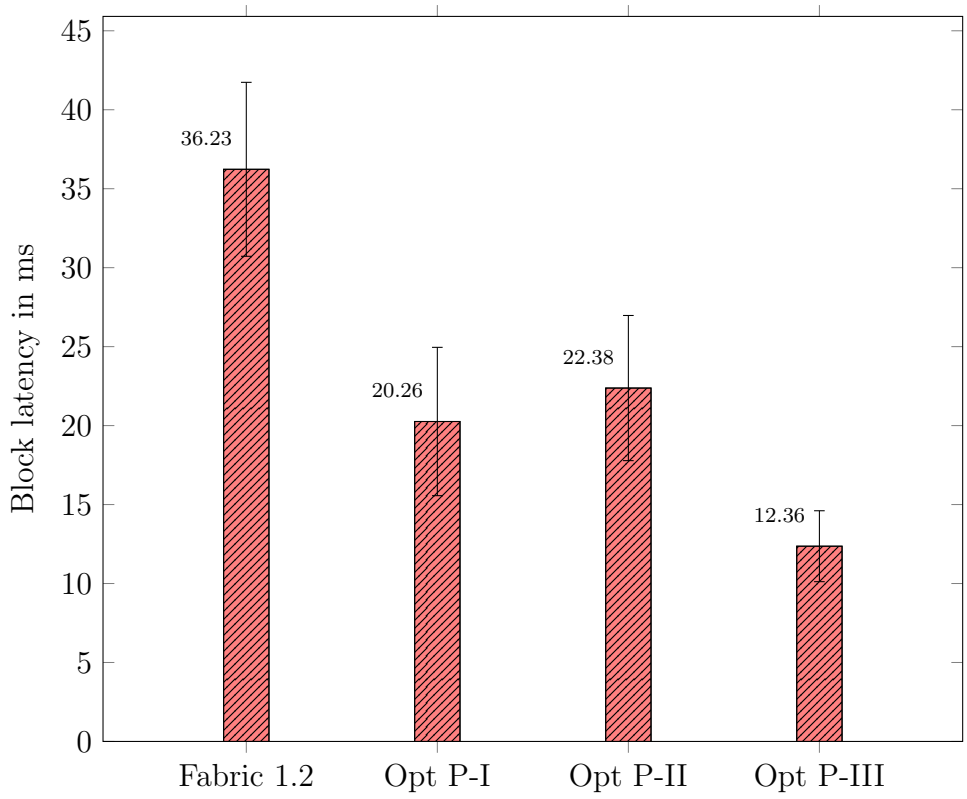


Figure 3.9: Impact of our optimizations on peer block latency.

Because of batching, we show the latency per block, rather than per-transaction latency. The results are in line with our self-imposed goal of introducing no additional latency as a result of increasing throughput; in fact our performance improvements decrease peer latency to a third of its original value (note that these experiments do not take network delay into account). Although the pipelining introduced in Opt P-II generates some additional latency, the other optimizations more than compensate for it.

By using a hash table for state storage (Opt P-I), we are able to more than double the throughput of a Fabric 1.2 peer from about 3,200 to more than 7,500 transactions/s. Parallelizing validation (Opt P-II) adds an improvement of roughly 2,000 transactions per second. This is because, as Figure 3.2 shows, only the first two validation steps can be parallelized and scaled out. Thus, the whole pipeline performance is governed by the throughput of read and write set validation and commitment. Although commitment is almost free when using Opt P-I, it is not until the introduction of the unmarshaling cache

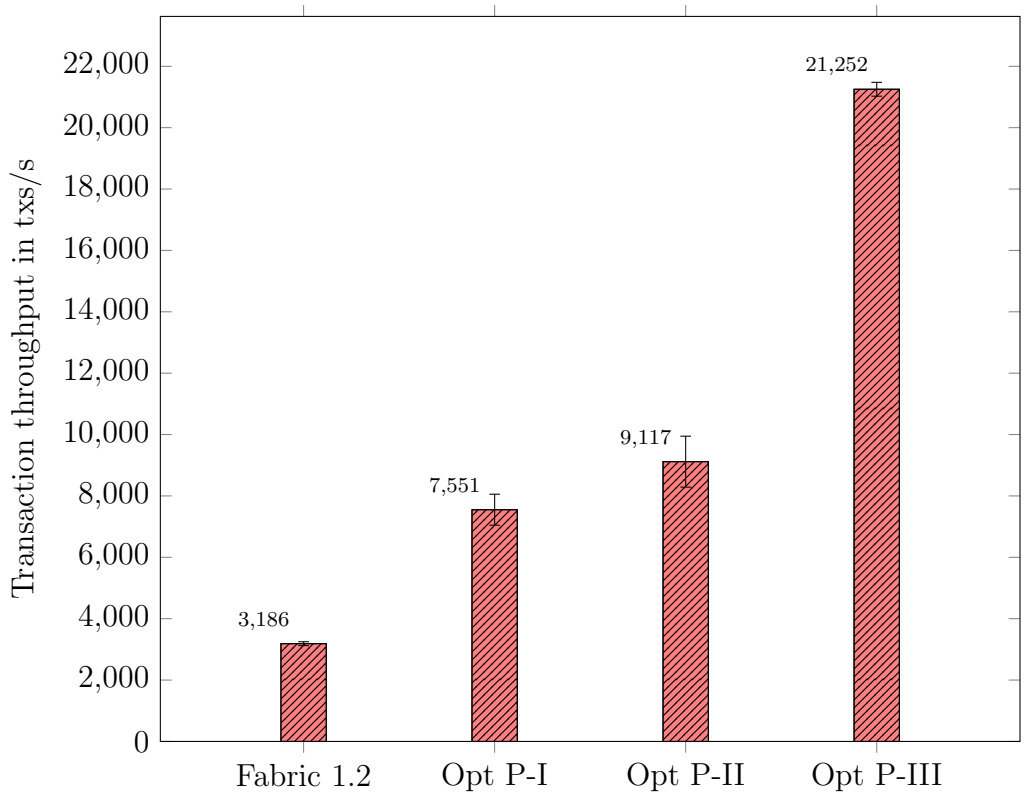


Figure 3.10: Impact of our optimizations on peer throughput.

in Opt P-III that Opt P-II pays off. The cache drastically reduces the amount of work for the CPU, freeing up resources to validate additional blocks in parallel. With all peer optimizations taken together, we increase a peer's commit performance by 7x from about 3,200 transactions/s to over 21,000 transactions/s.

Parameter sensitivity

As discussed in Section 3.5.3, parallelizing block and transaction validation at the peer is critical. However, it is not clear how much parallelism is necessary to maximize performance. Hence, we explore the degree to which a peer's performance can be tuned by varying two parameters:

- The number of go-routines concurrently shepherding blocks in the validation pipeline

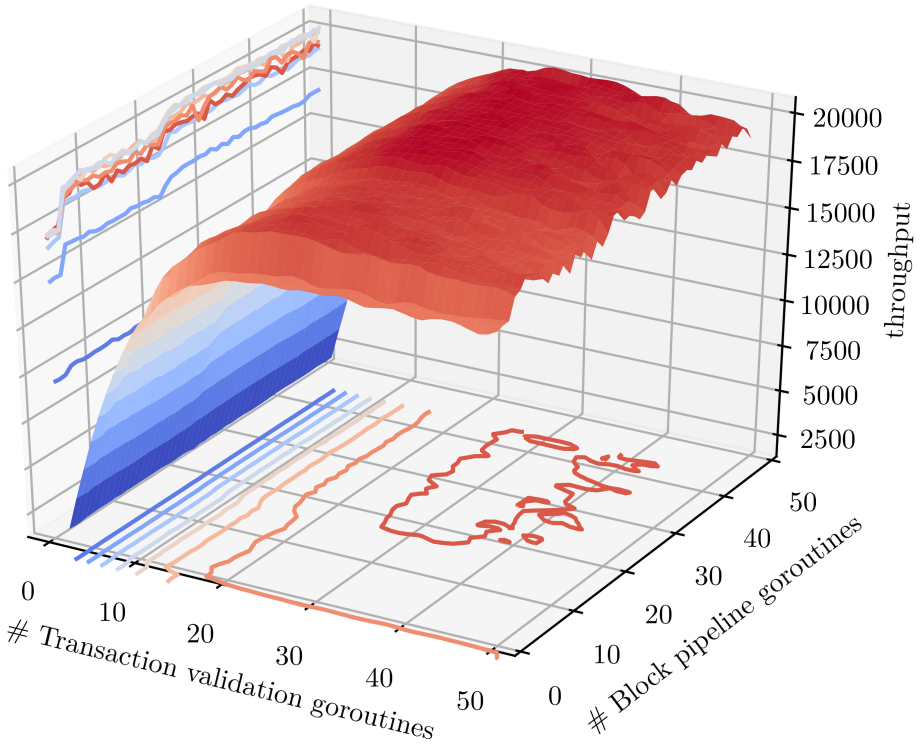


Figure 3.11: Parameter sensitivity study for blocks containing 100 transactions and a server with 24 CPU cores. We scale the number of blocks that are validated in parallel and the number of transactions per block that are validated in parallel independently.

- The number of go-routines concurrently validating transactions

We controlled the number of active go-routines in the system using semaphores, while allowing multiple blocks to concurrently enter the validation pipeline. This allows us to control the level of parallelism in block header validation and transaction validation through two separate go-routine pools.

For a block size of 100 transactions, Figure 3.11 shows the throughput when varying the number of go-routines. The total number of threads in the validation pipeline is given by the sum of the two independent axes. For example, we achieve maximum throughput for 25 transaction validation go-routines and 31 concurrent blocks in the pipeline, totalling 56 go-routines for the pipeline. While we see a small performance degradation through thread management overhead when there are too many threads, the penalty for starving

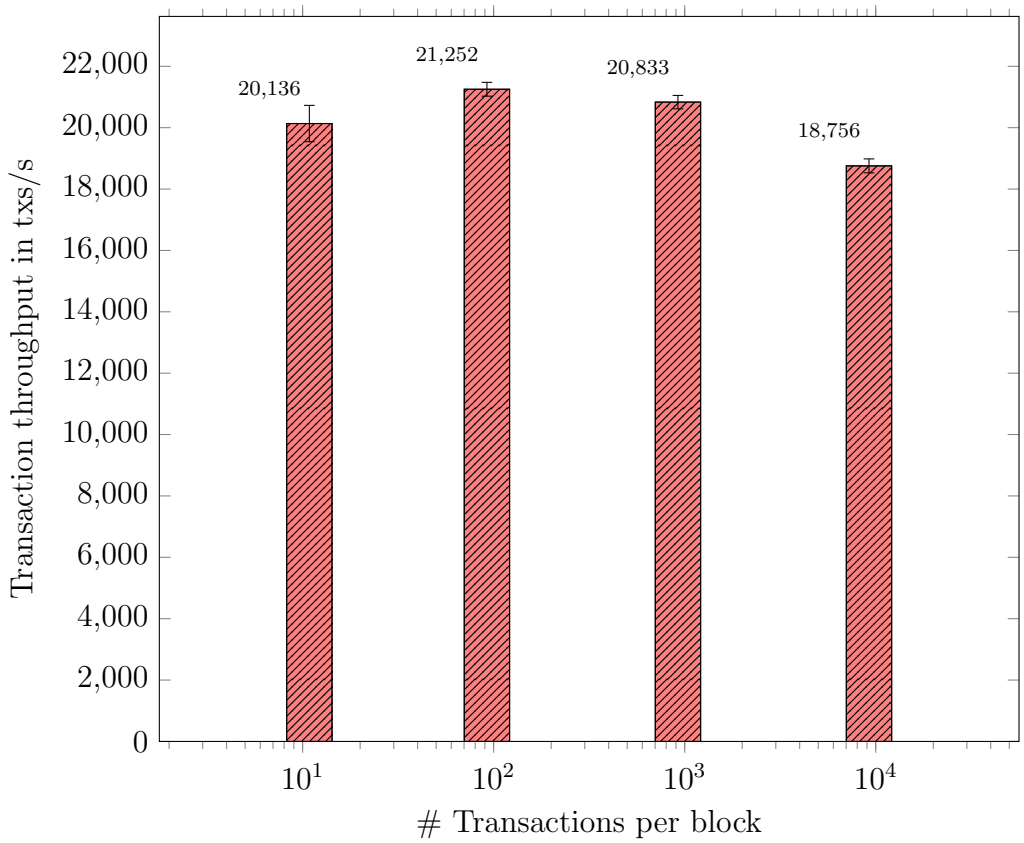


Figure 3.12: Throughput dependence on block size for optimally tuned configuration.

the CPU with too few parallel executions is drastic. Therefore, we suggest as a default that there be at least twice as many go-routines as there are physical threads in a given machine.

We now investigate the dependence of peer throughput on block size. Each block size experiment was performed with the best tuned go-routine parameters from the previous test. All configurations used around 24 ± 2 transaction validation go-routines and 30 ± 3 blocks in the pipeline. Again, we split 100,000 transactions among blocks of a given size for a single benchmark run and repeated the experiment 1,000 times. We chose to scan the block size space on a logarithmic scale to get an overview of a wide spectrum.

The results are shown in Figure 3.12. We find that a block size of 100 transactions/block gives the best throughput with just over 21,000 transactions per second. We also investigated small deviations from this block size. We found that performance differences for

block sizes between 50 and 500 were very minor, so we opted to fix the block size to 100 transactions.

Lastly, we compare the load distribution on the peer after all improvements are in place with the distribution of the default peer. An updated call graph is shown in Figure 3.13. First, note that the ratio of samples landing on the critical path rises to 66.5% because we moved endorsement and storage to external servers. The percentages of the major contributors to the critical path execution time after normalization are shown in Table 3.2. They still make up 70.44% of the computation on the critical path, so we can be sure that we did not miss another hidden bottleneck. Most of the time is now spent on parallel cryptographic computations. We have not altered the cryptographic computation apart from optimizing its parallelization. The same amount of signature and hash checks need to be performed to move a block through the pipeline from validation to commitment. Therefore, we can use its impact on the critical path to calibrate the improvements in database access and Protobuffer unmarshaling. We find that the impact of the database access has been reduced by a factor of 16 and the impact of Protobuffer unmarshaling by a factor of 10.

Table 3.2: Percentages of the overall execution time of the critical path on the peer after all improvements are included.

Call graph nodes	Critical path samples (%)
Cryptographic operations	44.33
Protobuffer unmarshaling	11.51
Hash table access	7.84
Memory allocation	6.76
	70.44

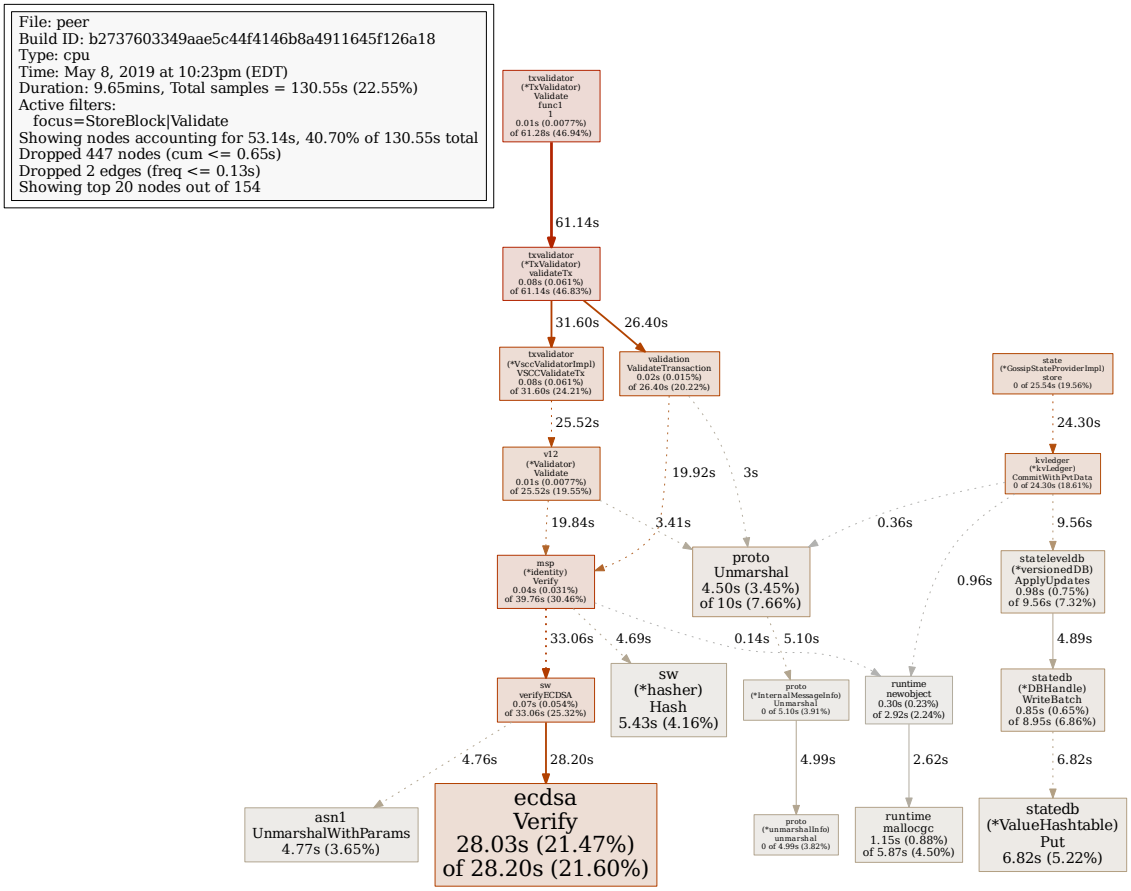


Figure 3.13: Call graph of the fully optimized peer. Execution time is dominated by cryptographic computations and memory allocation.

3.5.4 End-to-end throughput

We now discuss the end-to-end throughput achieved by combining all of our optimizations, i.e., Opt. O-II combined with Opt. P-III, compared to our measurements of unmodified Fabric 1.2.

We set up a single orderer that uses a cluster of three ZooKeeper servers and three Kafka servers, with the default topic replication factor of three, and connect it to a peer. Blocks from this peer are sent to a single data storage server that stores world state in LevelDB and blocks in the file system. For scale-out, five endorsers replicate the peer state and provide sufficient throughput to deal with client endorsement load. Finally, a client is installed on its own server; this client requests endorsements from the five endorser servers and sends endorsed transactions to the ordering service. This uses a total of fifteen servers connected to the same 1 Gbit/s switch in our local data center.

We send a total of 100,000 endorsed transactions from the client to the orderer, which batches them to blocks of size 100 and delivers them to the peer. To estimate throughput, we measure the time between committed blocks on the peer and take the mean over a single run. These runs are repeated 100 times. Table 3.3 shows a significant improvement of 6-7× compared to our baseline Fabric 1.2 benchmark.

Table 3.3: End-to-end throughput

	Fabric 1.2	FastFabric
Transactions/s	3185 ± 62	19112 ± 811

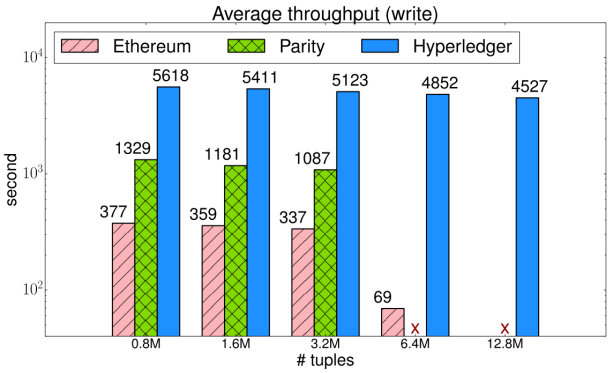
3.6 Related Work

While scalability is a hot research topic with ideas ranging from multi-chain solutions [118] to off-chain channels for micro-transactions between recurring trading partners [90], little research has been done into reliable performance testing of blockchain solutions. As of this writing, Dinh et al. [31] are to the best of our knowledge the only ones proposing a standardized benchmarking framework, albeit specifically for private blockchains. They identify four layers that together comprise a blockchain: *application, execution engine, data model* and *consensus*. They in turn propose several benchmarks that target specific layers. However, all benchmarks are still executed as end to end tests, so that these layers never are fully decoupled. This means, that even the IOHeavy benchmark workload in Figure 3.14 obscures a detailed view on the internal data management mechanisms, which is needed to make decisions about an optimized database design.

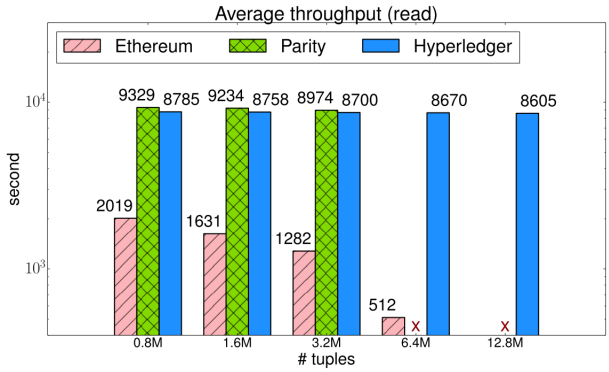
It has been shown that blockchain systems are still slower than traditional database systems [22, 102]. This led to to the exploration of introducing some blockchain elements into conventional database systems. Instead of trying to replace applications where currently a more traditional distributed database was in place with a blockchain, BigchainDB enriches the database with blockchain features like immutability and decentralized control. However, BigchainDB has since pivoted to a standard blockchain approach and discarded the original idea [41]. Now, it uses the Tendermint consensus protocol [15] to coordinate transactions across a cluster of MongoDB instances.

Hyperledger Fabric is a recent system that is still undergoing rapid development and significant changes in its architecture. Hence, there is relatively little work on the performance analysis of the system or suggestions for architectural improvements. Here, we survey recent work on techniques to improve the performance of Fabric.

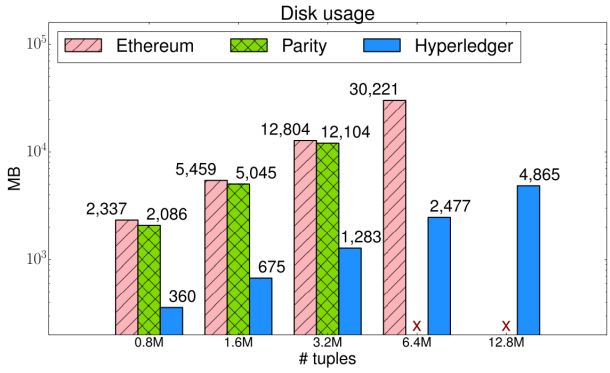
The work closest to ours is by Thakkar et al. [113] who study the impact of various configuration parameters on the performance of Fabric. They find that the major bottlenecks are repeated validation of X.509 certificates during endorsement policy verification, sequential policy validation of transactions in a block, and state validation during the commit phase. They introduce aggressive caching of verified endorsement certificates (incorporated into Fabric version 1.1, hence part of our evaluation), parallel verification of endorsement policies, and batched state validation and commitment. These improvements increase overall throughput by a factor of 16. We also parallelize verification at the committers and go one step further in replacing the state database with a more efficient data structure, a hash table.



(a) Write



(b) Read



(c) Disk usage

Figure 3.14: BLOCKBENCH[31]: IOHeavy workload, **'X'** indicates Out-of-Memory error

It is well known that Fabric's orderer component can be a performance bottleneck due to the message communication overhead of Byzantine fault tolerant (BFT) consensus protocols. Therefore, it is important to use an efficient implementation of a BFT protocol in the orderer. Bessani, Sousa et al. [12, 108] study the use of the well-known BFT-SMART implementation as a part of Fabric and show that, using this implementation within a single data center, a throughput of up to 30,000 transactions/second is achievable. However, unlike our work, the committer component is not benchmarked and the end-to-end performance is not addressed.

Androulaki et al. [4] study the use of channels for scaling Fabric. They use channels to shard the key space of the blockchain world state. This means that transactions to different channels are independent by design and can be completely parallelized. However, this work does not present a performance evaluation to quantitatively establish the benefits from their approach.

Raman et al. [96] study the use of lossy compression to reduce the communication cost of sharing state between Fabric endorsers and validators when a blockchain is used for storing intermediate results arising from the analysis of large datasets. However, their approach is only applicable to scenarios which are insensitive to lossy compression, which is not the general case for blockchain-based applications.

Some studies have examined the performance of Fabric without suggesting internal architectural changes to the underlying system. For example, Dinh et al. use BlockBench [31], a tool to study the performance of private blockchains, to study the performance of Fabric, comparing it with that of Ethereum and Parity. They found that the version of Fabric they studied did not scale beyond 16 nodes due to congestion in the message channel. Nasir et al. [80] compare the performance of Fabric 0.6 and 1.0, finding, unsurprisingly, that the 1.0 version outperforms the 0.6 version. Baliga et al. [5] showed that application-level parameters such as the read-write set size of the transaction and chaincode and event payload sizes significantly impact transaction latency. Similarly, Pongnumkul et al. [89] compare the performance of Fabric and Ethereum for a cryptocurrency workload, finding that Fabric outperforms Ethereum in all metrics. Bergman [10] compares the performance of Fabric to Apache Cassandra [64] in similar environments and finds that, for a small number of peer nodes, Fabric has a lower latency for linearizable transactions in read-heavy workloads than Cassandra. However, with a larger number of nodes, or write-heavy workloads, Cassandra has better performance.

3.7 Limitations and Future Work

In this work, we showed how we can dramatically increase the throughput of Hyperledger Fabric by consequently separating the tasks of transaction endorsement, validation and storage, moving them to independent hardware and taking advantage of the freed up resources by parallelizing the validation and commitment pipeline.

Our solution takes advantage of Fabric's ability to endorse transactions concurrently before they are ordered, because this avoids the bottleneck of sequential transaction execution. However, this makes Fabric susceptible to transaction conflicts since only the results are serialized. Thus, a large number of transactions could be discarded during validation because they try to modify the same keys. This makes the effective throughput dependent on the specific transaction workload. We will address this shortcoming in the next chapter.

Our final analysis clearly showed that with all optimizations in place the cryptographic operations have become the new bottleneck. Our initial investigations pointed towards the big integer library that is used internally by Golang's standard crypto library as having the biggest impact on performance. This could potentially be replaced by a more efficient implementation. At that point, we expect the ordering service to become the bottleneck of the blockchain system again. Thus, new fast consensus protocols like RCanopus [60] and Mir-BFT [110] should be explored as alternatives to existing solutions like Kafka and Raft.

By splitting Fabric's peer across multiple servers, we increase the risk of faulty behaviour. Not only does a hardware fault become more likely with each additional server, it also extends the overall attack surface that a malicious attacker can exploit. However, these servers should be very localized, i.e. installed in the same rack in a data center, so adding more hardware should not dramatically change the probability of a successful attack. Furthermore, data centers have decades of experience in dealing with hardware failures, so we believe the benefits by far outweigh the risks. Therefore, future research could explore the possibility of scaling out even further and spreading the validation across multiple servers as well.

While we address the challenge of generating enough transactions concurrently to use the system to full capacity by scaling endorsers independently from their validating fast peer, our preliminary experiments showed that there is also the potential to make the endorsement step more efficient. Because chaincode execution happens in an isolated Docker container, each interaction with the world state requires the serialization and deserialization of data and the use of remote procedure calls. We believe that the endorser can be sped up with similar techniques we implemented on the fast peer like minimizing serializa-

tion. Moreover, the idea to move a copy of the world state into the container itself to get rid of the remote calls could be explored.

Lastly, the decoupled storage server currently only persists a local copy of the ledger and world state. With all the ressources previously reserved for endorsement and validation, we can envision building this out to a more sophisticated data analytics system.

Chapter 4

XOX Fabric: Dealing with skewed workloads

In the previous chapter, we have shown a modified Fabric can handle tens of thousands of transactions per second. However, this performance is only achievable for contention-free transaction workloads. If many transactions compete for a small set of hot keys in the world state, the effective throughput drops drastically. We therefore propose XOX: a novel two-pronged transaction execution approach that both minimizes invalid transactions in the Fabric blockchain and maximizes concurrent execution. Our approach additionally prevents unintentional denial of service attacks by clients re-submitting conflicting transactions. Even under *fully* contentious workloads, XOX can handle more than 3,000 transactions per second, *all* of which would be discarded by regular Fabric. As we will show in Section 4.2 we preserve Fabric's modularity completely.

4.1 The Hot Key Theorem

We now state and prove a theorem that limits the performance of any OX system.

Hot Key Theorem. *Let $\bar{l}$ be the average time between a transaction's execution and its state transition commitment. Then the average effective throughput for all transactions operating on the same key is at most $\frac{1}{\bar{l}}$.*

Proof. The proof is by induction. Let i denote the number of changes to an arbitrary but fixed key k.

$i = 0$ (just before the first change):

For k to exist, there must be exactly one transaction tx_0 which takes time l_0 from execution to commitment and creates k with version v_1 at time t_1.

$i \to i + 1$ (just before the $i + 1^{th}$ change):

Let k's current version be v_i at time t_i. Let tx_i be the first transaction in a block which updates k to a new version v_{i+1}. The version of k during tx_i's execution must have been v_i, otherwise Fabric would invalidate tx_i and prevent commitment. Let tx_i be committed at time t_{i+1} and l_i be the time between tx_i's execution and commitment. Therefore,

$$t_i \leq t_{i+1} - l_i.$$

Likewise, no transaction tx'_i which is ordered after tx_i can commit an update $v_i \to v'_{i+1}$ because tx_i already changed the state and tx'_i would therefore be invalid. Consequently, tx_i must be the only transaction able to update k from v_i to a newer version.

This means, N updates to k take t_N time with

$$t_N \geq \sum_{i=0}^{N-1} l_i.$$

A lower bound on the average update time is then given by

$$\frac{1}{N} t_N \geq \sum_{i=0}^{N-1} \frac{1}{N} l_i = \bar{l},$$

so we get $\frac{1}{\bar{l}}$ as an upper bound on throughput being the inverse of the update latency. □

This theorem has a crucial consequence. For example, FastFabric can achieve a nominal throughput of up to 20,000 transactions per second, yet even an unreasonably fast transaction life cycle of 50 ms from execution to commitment would result in a maximum of 20 updates per second to the same key, or once every ten blocks with a block size of 100 transactions. Worse yet, transactions are not only invalidated if their RW set overlaps completely, but also if there is a *single* key overlap with a previous transaction. This means that workloads with hot keys can easily reduce effective throughput by several orders of magnitude.

While early abort schemes can discard invalid transactions before they become part of a block, they cannot break the theorem. Assuming they result in blocks without invalid

transactions, they can only fill up the slots in a new block with transactions using different key spaces. Thus, they skew the processed transaction distribution. Furthermore, aborted transactions need to be re-submitted and re-executed, flooding the network with even more attempts to modify hot keys. Eventually, endorsers will be completely saturated by clients repeatedly trying to get their invalid transactions re-executed.

4.2 The XOX hybrid model

To deal with the drawbacks of both the OX and XO patterns, we now present the *execute-order-execute (XOX)* pattern which adds a secondary post-order execution step to execute the patch-up code added to smart contracts. XOX minimizes transaction conflicts while preserving concurrent block processing and without the introduction of any centralized elements. In this section, we first describe the necessary changes to the endorsers' pre-order execution step to allow the inclusion of external oracles in the post-order execution step. Then, we describe changes to the critical transaction flow path on the peers after they receive blocks from the ordering service. The details of the crucial steps we introduce are described in Sections 4.3 and 4.4. Notably, our changes do not affect the ordering service, preserving Fabric's modular structure.

4.2.1 Pre-order endorser execution

The pre-order execution step leverages concurrent transaction execution and uses general purpose programming languages like *Go*. Depending on the endorsement policy, clients request multiple endorsers to execute their transaction and the returned execution results must be identical. This makes a deterministic execution environment unnecessary because deviations are discarded and a unanimous result from all endorsers becomes ground truth for the whole network. Notably, this also allows external oracles like weather or financial data. If these oracle data lead to non-deterministic RW sets, the client will not receive identical endorser responses and the transaction will never reach the Fabric network.

External oracles are a powerful tool. If they are supported by the pre-order execution step, they must also be supported by the post-order execution step. To achieve this, we must make the oracle deterministic. We leverage the same mechanism that ensures deterministic transaction results for pre-order execution: We extend the transaction response by an additional *oracle set*. Any external data are recorded in the form of key-value pairs and are added to the response to the client. Now, if the oracle sets for the same transaction

executed by different endorsers differ, the client has to discard the transaction. Otherwise, the external data effectively becomes part of the deterministic world state, so that it can be used by the post-order execution step without jeopardizing consistency. Analogous to existing calls to `GetState` and `PutState` that record the read and write set key-value pairs, respectively, we add a new call `PutOracle` to the chaincode API.

4.2.2 Critical transaction flow path

Our previous work on FastFabric showed how to improve performance by pipelining the syntactic block verification and endorsement policy validation (EP validation) so that it can be done for multiple blocks at the same time. However, the RW set validation to check for invalid state transitions and the final commitment had to be done sequentially in a single thread. While the the XOX model is an orthogonal optimization, its second execution step needs to be placed between RW set validation and commitment. Since this step is relatively slow, we must expand our concurrency efforts to pipelining RW set validation, post-order executions, and commitment. Two vital pieces for this effort, a transaction dependency analyzer and the executions step itself, are described in later sections in detail, so we will only give a brief overview here. This allows us to concentrate on the pipeline integration in this section.

Dependency analyzer

For concurrent transaction processing, we rely on the ability to isolate transactions from each other. However, the sequential order of transactions in a block matters when their RW sets are validated and they are committed. A dependency exists when two transactions overlap in some keys of their RW sets (read-only transactions do not even enter the orderer). In that case, we cannot process them independently. Therefore, we need to keep track of dependencies between transactions so we know which subsets of transactions can be processed concurrently.

Execution step

Transactions for which the dependency analyzer has found a dependency on an earlier transaction would be invalidated during Fabric's RW set validation. We introduce a step which re-executes transaction with such an RW set conflict based on the most up-to-date

world state. It can resolve conflicts due to a lack of knowledge of concurrent transactions during pre-order execution. However, it still invalidates transactions that attempt something the smart contract does not allow, such as creating a negative account balance.

In FastFabric, peers receive blocks as fast as the ordering service can deliver them. If the syntactic verification of a block fails, the whole block is discarded. Thus, it is reasonable to keep this as a first step in the pipeline. Next up is the EP validation step. Each transaction can be validated in parallel because the validations are independent of each other. The next step is the intertwined RW set validation and commitment: Each transaction is validated, and, if successful, added to an update batch that is subsequently committed to the world state.

XOX Fabric separates RW set validation from the commitment decision. Therefore, this step is no longer dependent on the result of the EP validation and can be done in parallel. However, in order to validate transactions concurrently, we need to know their dependencies, so the dependency analyzer goes first and releases transactions to the RW set validation as their dependencies are resolved.

Subsequently, the results from the EP validation and RW set validation are collected, and if they are marked as valid, they can be committed concurrently. If a RW set conflict arises, they need to be sent to the new execution step to be re-executed based on the current world state. Finally, successfully re-executed transactions are committed and all others are discarded.

Our design allows dependency analysis to work in parallel to endorsement policy validation and transactions can proceed as soon as all previous dependencies are known. Specifically, independent sets of transactions can pass through RW set validation, post-order execution, and commitment steps concurrently. The modified pipeline is shown in Figure 4.1.

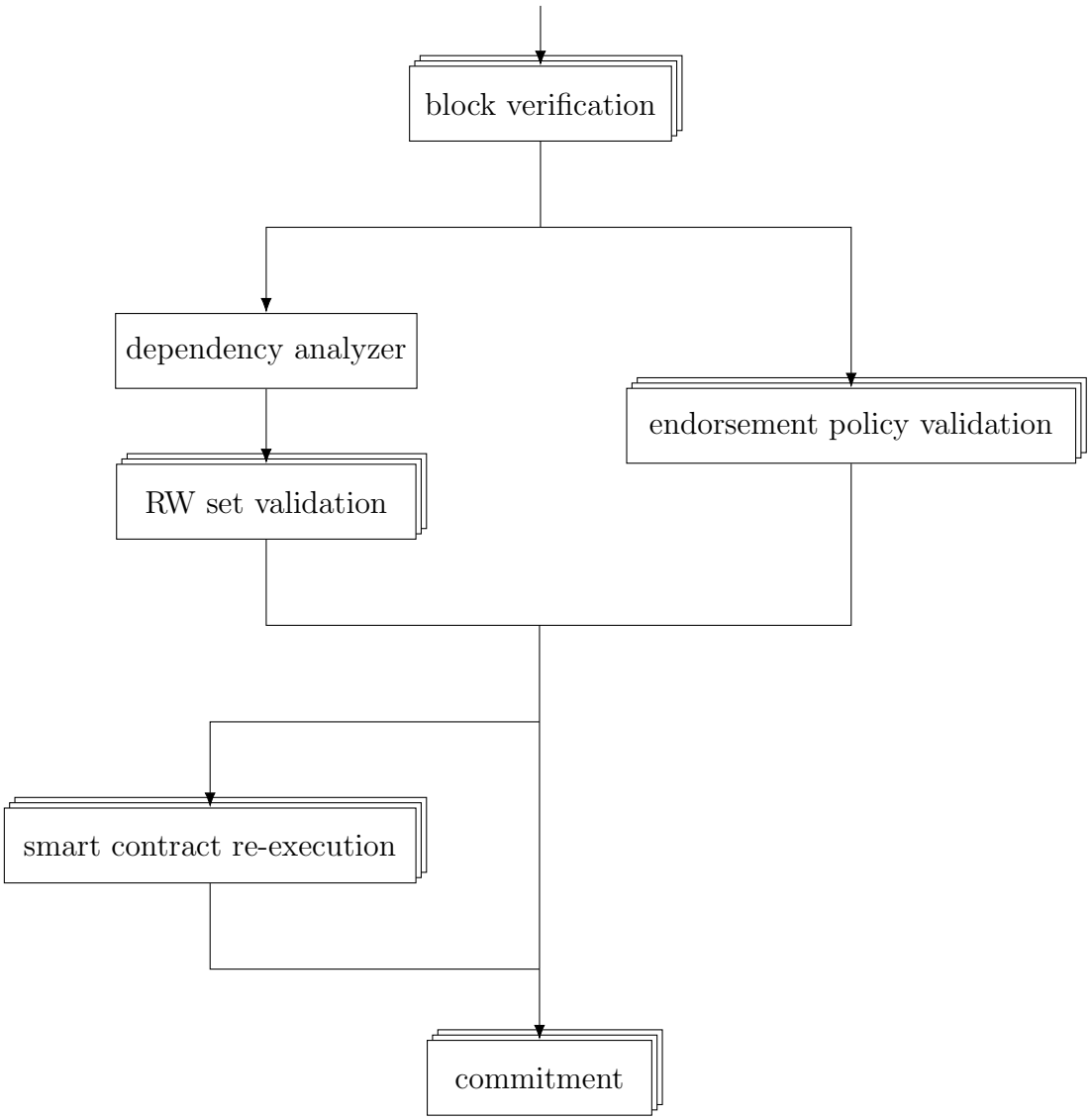


Figure 4.1: The modified XOX Fabric validation and commitment pipeline. Stacks and branched paths show parallel execution.

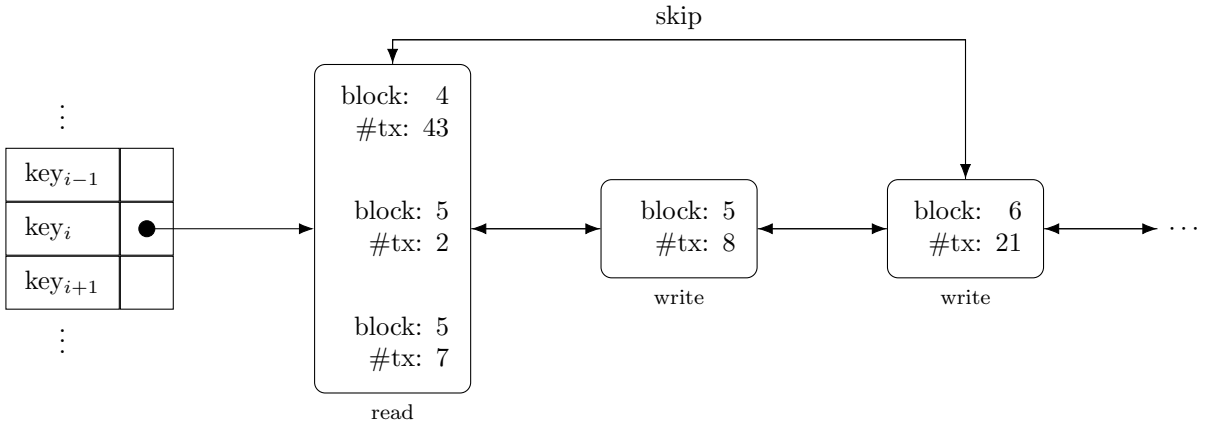


Figure 4.2: Dependency analyzer data structure: Example of a state database key mapped to a doubly-linked skiplist of dependent transactions.

4.3 Dependency analyzer

We now discuss the details of the dependency analyzer. Note that the only way for a transaction to have a dependency on another is an overlap in its RW set with a previous transaction. More precisely, one of the conflicting transactions must write to the overlapping key. Reads do neither change the version nor the value of a key, so they do not impede each other. However, we must consider a write a blocking operation for that key. If transaction a with a write is ordered before transaction b with a read from the same key, then this must always happen in this order lest we lose deterministic behaviour of the peer because of the changing key value. The reverse case of read-before-write has the same constraints. In the write-write case, neither transaction actually relies on the version or the value of that key. Nevertheless, they must remain in the same order, otherwise transaction a's value might win out, even though transaction b should overwrite it.

To detect such conflicts, we keep track of read and write accesses to all keys across transactions. For each key, we create a doubly-linked skip list that acts as a dependency queue, recording all transactions that need to access it. Entries in this queue are sorted by the blockchain transaction order. As described before, consecutive reads of the same key do not affect each other and can be collapsed into a single node in the linked list so they will be freed together. For faster traversal during insertion, nodes can skip to the start of the next block in the list. This data structure is illustrated in Figure 4.2. After the analysis of a transaction is complete, it will not continue to the next step in the pipeline until all previous transactions have also been analyzed, lest an existing dependency might

be missed.

Dependencies may change in two situations: when new transactions are added or existing transactions have completed the commitment pipeline. In either case, we update the dependency lists accordingly and check the first node of lists that have been changed. If any of these transactions have no dependency in any key anymore, they are released into the validation pipeline. However, we can only remove a transaction from the dependency lists once it is either committed or discarded, lest dependent transactions get freed up prematurely.

4.4 Post-order execution step

The post-order execution step executes additional patch-up code added to a smart contract. We discuss it in more detail in this section.

When the RW validation finds a conflict between a transaction's RW set and the world state, that transaction will be re-executed and possibly salvaged using the patch-up code. However, the post-order execution stage needs to adhere to some constraints. First, the new RW set must be a subset of the original RW set so the dependency analyzer can reason properly. Without this restriction, new dependencies could emerge and transactions scheduled for parallel processing would now create an invalid world state. Second, the blockchain network also needs consistency among peers. Therefore, the post-order execution must be deterministic so there is no need for further consensus between peers. Lastly, this new execution step is part of the critical path and thus should be as fast as possible.

For easier adoption of smart contracts from other blockchain systems, we use a modified version of Ethereum's EVM [117] as the re-execution engine for patch-up code[1]. Patch-up code take a transaction's read set and oracle set as input. The read set is used to get the current key values from the latest version of the world state. Based on this and the oracle set, the smart contract then performs the necessary computations to generate a new write set. If the transaction is not allowed by the logic of the smart contract based on the updated values, it is discarded. Finally, in case of success, it generates an updated RW set, which is then compared to the old one. If all the keys are a subset of the old RW set, the result is valid and can be committed to the world state and blockchain.

For example, suppose client A wants to add 70 digital coins to an account with a current balance of 20 coins. Simultaneously, client B wants to add 50 coins to the same account.

[1]We note that forays have been made to build WebAssembly based execution engines [36], which would allow for a variety of programming languages to build smart contracts for the post-order execution step.

They both have to read the key of the account, update its value, and write the new value back, so the account's key is in both transactions' RW set. Even if both clients are honest, only the transaction which is ordered earlier will be committed. Without loss of generality, assume that A's transaction updates the balance to 90 coins because it won the race. In XOX Fabric, B's transaction would wait for A to finish due to its dependency and then would find a key version conflict in the RW validation step. Therefore, it is sent to the post-order execution step. In the step, B's patch-up code can read the updated value from the database and add its own value for a total of 140 coins, which is recorded in its write set. After successful execution, the RW set comparison is performed and the new total will be committed. Thus, the re-execution of the patch-up code salvages conflicting transactions.

However, if we start with an account balance of 100 coins and A tries to subtract 50 coins and B tries to subtract 60 coins, we get a different result. Again, B's transaction would be sent to be re-executed. But this time, it's patch-up code tries to subtract 60 coins from the updated 50 coins and the smart contract does not allow a negative balance. Therefore, B's transaction will be discarded, even though it was re-executed based on the current world state.

Thus, our hybrid XOX approach can correct transactions which would have been discarded because they were executed based on a stale world state. However, transactions that do not satisfy the smart contract logic are still rejected.

Lastly, if we do not put any restrictions on the execution, we risk expensive computations, low throughput, and even non-terminating smart contracts. Ethereum deals with this by introducing *gas*. If a smart contract runs out of gas, it is aborted and the transaction is discarded. As of yet, Fabric does not include such a concept.

As a solution, we introduce *virtual gas* as a tuning parameter for system performance. Instead of originating from a bid by the client that proposes the transaction, it can be set by a system administrator. If the post-order step runs out of gas for a transaction, it becomes immediately invalidated, but in case of success the fee is never actually paid. A larger value allows for more complex computation at the cost of throughput. While the gas parameter should generally be as small as possible, large values could make sense for workloads with very infrequent transaction conflicts and high importance of conflict resolution.

4.5 Experiments

We now evaluate the peformance of XOX Fabric. We used 11 local servers connected by a 1 Gbit/s switch. Each is equipped with two Intel® Xeon® CPU E5-2620 v2 processors at 2.10 GHz, for a total of 24 hardware threads and 64 GB of RAM. We compare three systems with different capabilities. Fabric 1.4 is the baseline. Next, FastFabric adds efficient data structures, improved parallelization, and decoupled endorsement and storage servers. Finally, our implementation of an XOX model based on FastFabric adds transaction dependency analysis, concurrent key version validation, and transaction re-execution.

For comparable results, we match the network setup of all three systems as closely as possible. We use a single orderer in *solo* mode, ensuring that throughput is bound by the peer performance. A single anchor peer receives blocks from the orderer and broadcasts them to four endorsing peers. In the case of FastFabric and XOX, the broadcast includes the complete transaction validation metadata so endorsers can skip their own validation steps. FastFabric and XOX run an additional persistent storage server because in these cases the peers store their internal state database in-memory. The remaining four servers are used as clients[2]. Spawning a total of 200 concurrent threads, they use the Fabric node.js SDK to send transaction proposals to the endorsing peers and consecutively submit them to the orderer. Each block created by the orderer contains 100 transactions.

All experiments run the same chaincode: A money transfer from one account to another is simulated, reading from and writing to two keys in the state database, e.g. deducting 1 coin from *account0* and adding 1 coin to *account1*. We use the default endorsement policy of accepting a single endorser signature. XOX's second execution phase integrates a Python virtual stack machine (VM) implemented in Go [93]. We added a parameter to the VM to stop the stack machine after executing a certain amount of operations, emulating a *gas* equivalent. We load a Python implementation of the Go chaincode into the VM and extract the call parameters from the transaction so that the logic between pre-order and post-order execution remains the same. Therefore, the only semantic difference between XO and OX is that OX operates on up-to-date state.

For each tested system, clients generate a randomized load with a specific contention factor by flipping a loaded coin for each transaction. Depending on the outcome, they either choose a previously unused account pair or the pair *account0–account1* to create a RW set conflict. We scale the transaction contention factor from 0% to 100% in 10% steps and run the experiment for each of the three systems. Every time, clients generate a total of 1.5 Million transactions. In the following, we will discuss XOX's throughput

[2]We do not use Caliper because it is not sufficiently high-performance to fully load our system.

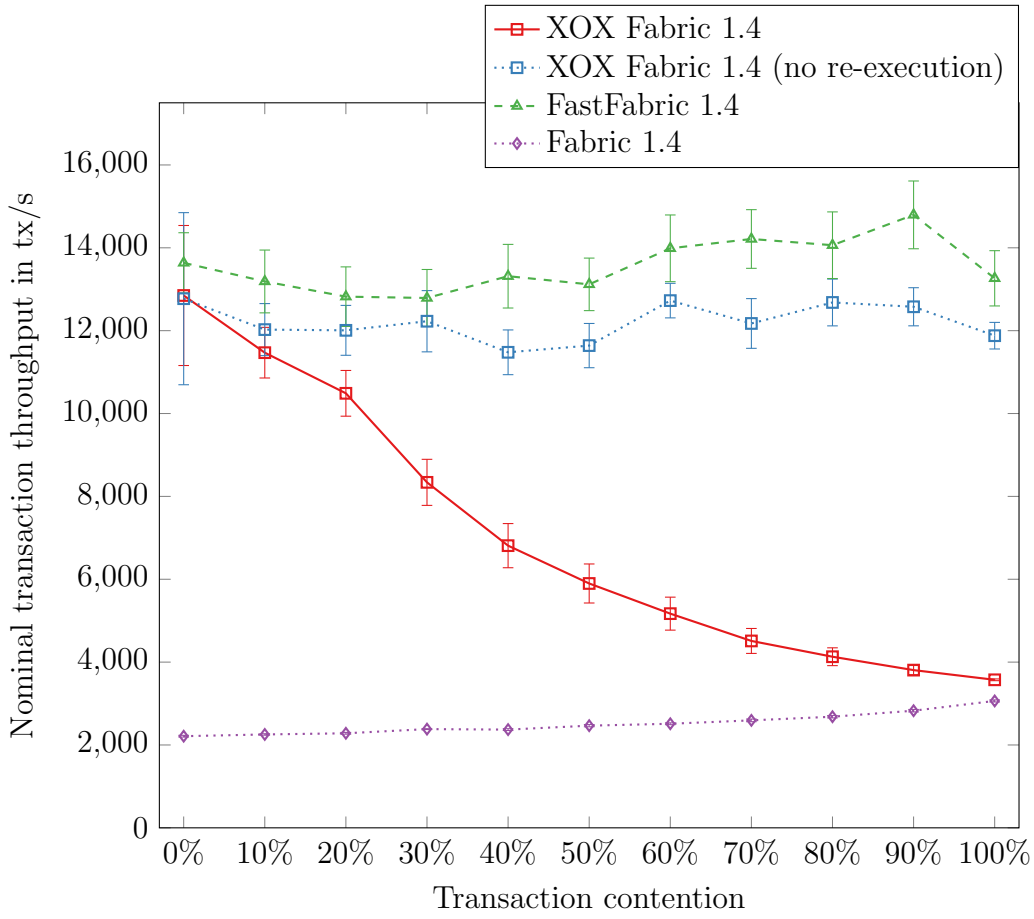


Figure 4.3: Impact of transaction conflicts on nominal throughput, counting both valid and invalid transactions.

improvements under contention over both FastFabric and Fabric 1.4, and its overhead compared to FastFabric.

4.5.1 Throughput

We start by examining the nominal throughput of each system in Figure 4.3. We measured the throughput of all transactions regardless of their validity. The effectively single-threaded validation/commitment pipeline of Fabric 1.4 creates results with little variance over time. The throughput increases slightly from about 2,200 tx/s to 3,000 tx/s the higher

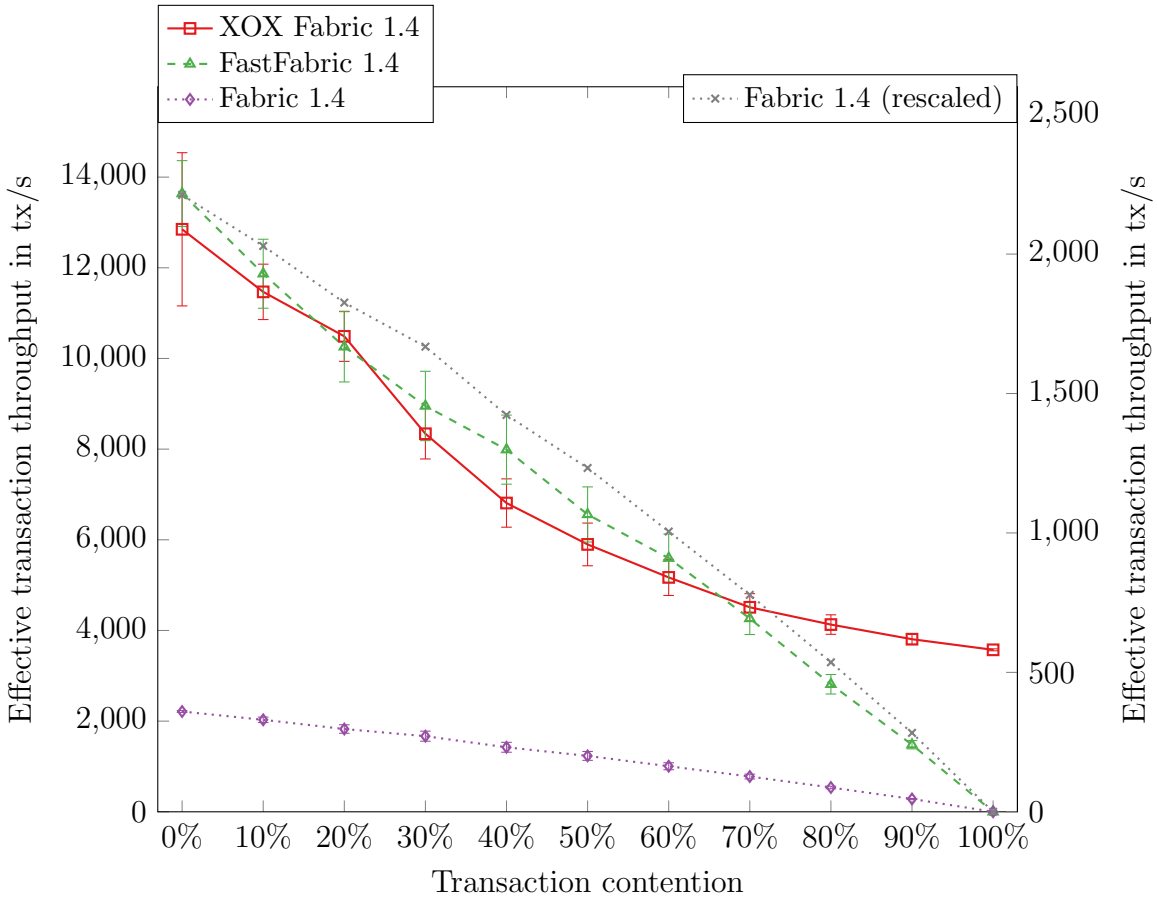


Figure 4.4: Impact of transaction conflicts on effective throughput, counting only valid transactions. Fabric 1.4 scaled up for slope comparison (right y-axis).

the transaction contention becomes, because Fabric discards invalid transactions, so their changes are not committed to the world state database. FastFabric follows the same trend, going from 13,600 tx/s up to 14,700 tx/s, although the relative throughput increase is not as pronounced because the database commit is cheaper, and there is higher variance due to many parallel threads vying for resources at times.

We ran the experiments for XOX in two configurations to understand the effects of different parts of our implementation on the overall throughput. First, we only included changes to the validation pipeline and the addition of the dependency analyzer but disabled transaction re-execution. Subsequently, we ran it again with all features enabled. The first configuration shows roughly the same behaviour as FastFabric, albeit with a small hit to

overall throughput, ranging from 12,000 tx/s up to 12,700 tx/s. For higher contention ratios, the fully-featured configuration's throughput drops from 12,800 tx/s to about 3,600 tx/s, a third of its initial value. However, this is expected as more transactions need to be re-executed sequentially. Importantly, even under full contention, XOX performs better than Fabric 1.4.

Note that the nominal throughout is meaningless if blocks contain mostly invalid transactions. Therefore, we now discuss the effective throughput. In Figure 4.4, we have eliminated all invalid transactions from the throughput measurements. Naturally, this means there is no change for the full XOX implementation, because it already produces only valid transactions. For better comparison of the three systems under contention, we normalized the projections of their plots. FastFabric and XOX follow the left y-axis while Fabric follows the right one. For up to medium transaction contention, all systems roughly follow the same slope. However, while both FastFabric and Fabric tend towards 0 tx/s in the limit of 100% contention, XOX still reaches a throughput of about 3,600 tx/s. At this point, all transaction in a submitted block have to be re-executed. This means, starting at 70% contention, XOX surpasses all other systems in terms of effective throughput while maintaining comparable throughput before that threshold.

Even though it might seem like a corner case, this is a significant improvement. All experiments were run with a synthetic static workload where the level of contention stayed constant. However, in a real world scenario, users have two options when their transaction fails. They can abandon the transaction, or, more likely, submit the same transaction again. In a system with some amount of contention, conflicting transactions can accumulate over time by users resubmitting them repeatedly. This results in an unintended denial of service attack. In contrast, XOX guarantees liveness in every scenario. What is more, even in the case where the number of conflicting transactions is not enough to have a snowball effect, the Hot Key Theorem still holds for impacted keys. So any transaction that accesses a hot key will always be at the 100% point of Figure 4.4, even if the vast majority of transactions is unaffected. This provides a massive push to the throughput of all transactions that deal with conflicts.

4.5.2 Overhead

We now explore the overhead of XOX compared to FastFabric's nominal performance in Figure 4.5. We isolate the overhead introduced by adding the dependency analyzer and modifying the validation pipeline so that it can handle single transactions instead of complete blocks, as well as the overhead of the transaction re-execution by the python VM.

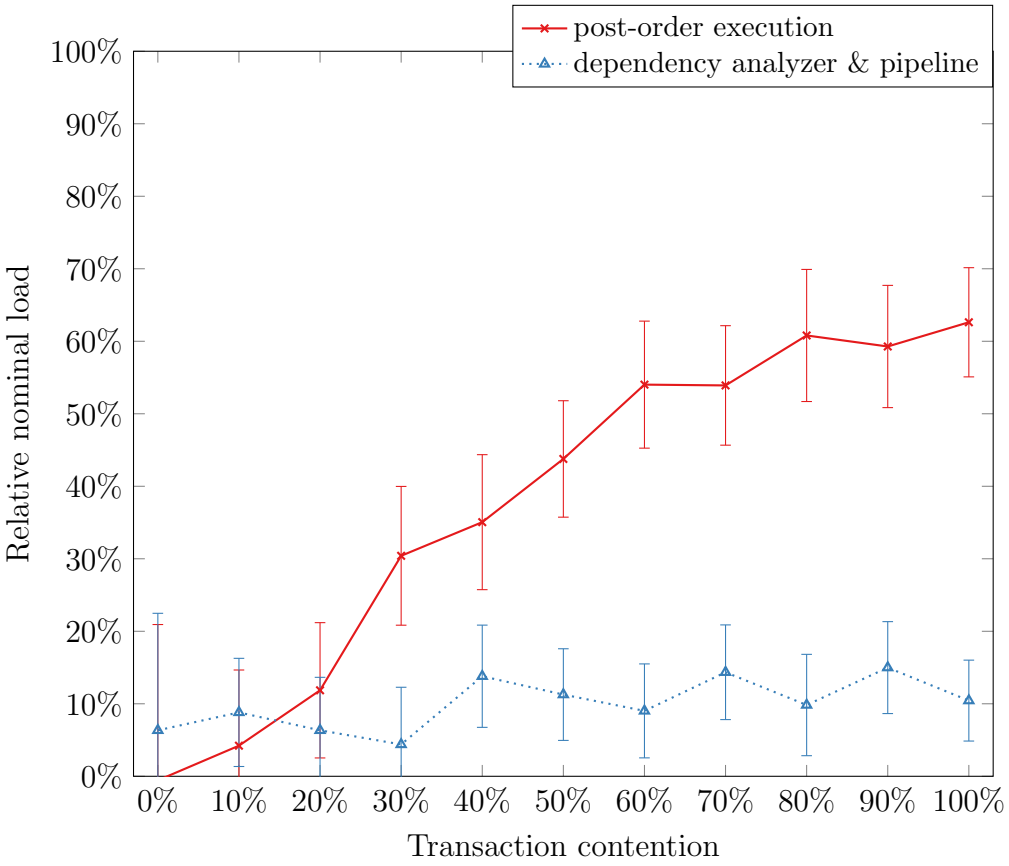


Figure 4.5: Relative load overhead of separate XOX parts over FastFabric.

The blue dashed line shows that the dependency analyzer overhead is almost constant regardless of contention level. By minimizing spots in the validation/commitment pipeline that require a sequential processing order of transactions, we achieve an overhead of less than 15% even when the dependency analyzer only releases one transaction at a time.

In contrast, the overhead of the re-execution step is more noticeable. For high contention, this step generates over 60% of additional load. Yet, this also means that replacing the highly inefficient Python VM used in our proof of concept with a faster deterministic execution environment could dramatically increase XOX's throughput for high-contention workloads. This would push the threshold when XOX beats the other systems to lower fractions of conflicting transactions. Furthermore, the contention load used for these experiments presents the absolute worst case, where every conflicting transaction is touching the same state keys, resulting in a fully sequential re-execution of all transactions. However, if

instead of a single account pair *account0–account1* used for contentious transactions there was a second pair *account2–account3*, the OX step would run in two concurrent threads instead of one. Even with this simple relaxation, the overhead would roughly be cut in half.

4.6 Related Work

Performance is an important issue for blockchain systems since they are still slower than traditional database systems [22, 31]. While most research focuses on consensus algorithms, less work has been done to optimize other aspects of the transaction flow, especially transaction execution.

In recent work, Sharma et al. [106] study the use of database techniques, i.e., transaction reordering and early abort, to improve the performance of Fabric. However, they do not follow its modular design and closely couple the different building blocks. For both early abort and transaction reordering, the ordering service needs to understand transaction contents to unpack and analyze RW sets. Furthermore, transaction reordering only works in pathological cases. Whenever a key appears both in the read and write set, which is the case for any application that transfers any kind of asset, reordering will not eliminate RW set conflicts. Some of their ideas related to early identification of conflicting transactions are orthogonal to ours and can be incorporated into our solution. However, some ideas, such as having the orderers drop conflicting transactions, are not compatible with our solution. We chose to keep to Fabric's design goal of allocating different tasks to different types of nodes, so our orderers do not examine the contents of the read and write sets.

Amiri et al. [2] introduce ParBlockchain using a similar architecture to Fabric's but with an OX model. Here, the ordering service also generates a dependency graph of the transactions in a block. Subsequently, transactions in the new block are distributed to nodes in the network to be executed, taking the dependencies into account. Only a subset of nodes executes any given transaction and shares the result with the rest of the network. This approach has two drawbacks. First, the ordering service must determine the transaction dependencies before they are executed. This requires the orderers to have complete knowledge of all installed smart contracts, and, as a result, restricts the complexity of allowed contracts. Even if a single conditional statement relies on a state value, for example *Read the value of key k, where k is the value to be read from key k'*, reasoning about the result becomes impossible. Second, depending on the workload, all nodes may have to communicate the current world state after every transaction execution to resolve execution deadlocks. This leads to a significant networking overhead.

We base the XOX work on *FastFabric*, our previous optimization of Hyperledger Fabric described in Chapter 3. We introduced efficient data structures, caching, and increased parallelization in the transaction validation pipeline to increase Fabric's throughput for *conflict-free* transaction workloads by a factor six to seven. In this work, we address the issue of conflicting transactions.

To the best of our knowledge, a document from the Fabric community [107] is the first to propose a secondary post-order execution step for Fabric. However, the allowed commands were restricted to addition, subtraction, and checking if a number is within a certain interval. Furthermore, this secondary execution step is always triggered regardless of the workload, and is not parallelized. This diminishes the value of retaining the first pre-order execution step and introduces the same bottleneck that OX models have to deal with.

Nasirifard et al. [81] take this idea one step further. By introducing conflict-free replicated data types (CRDTs), which allow conflicting transactions to be merged during the validation step, they follow a similar path to our work. However, their solution has several limitations. It can only process transactions sequentially, one block at a time. When conflicts are discovered, they use the inherent functionality of CRDTs to resolve them. While this enables efficient computation, it also restricts the kind of execution that can be done. For example, it is not possible to check a condition like negative account balance before merging two transaction results.

Zhang et al. [122] present a solution for a client-side early abort mechanism for Fabric. They introduce a transaction cache on the client that analyzes endorsed transactions to detect RW set conflicts and only sends conflict-free transactions to the ordering service. Transactions that have dependencies are held in the cache until the conflict is resolved and then they are sent back to the endorsers for re-execution. This approach prevents invalid transactions from a single client, but cannot deal with conflicts between multiple clients. Moreover, it cannot deal with hot key workloads.

Lastly, Escobar et al. [33] investigate parallel state machine replication. They focus on efficient data structures to keep track of parallelizable, i.e., independent state transitions. While this might be interesting to incorporate into Fabric in the future, we show in Section 4.5 that the overhead of our relatively simple implementation of a dependency tracker is negligible compared to the transaction execution overhead.

4.7 Limitations and Future Work

XOX mitigates Hyperledger Fabric's problem with producing invalid transactions when the workload focuses on a highly contested key space by detecting transaction conflicts early in the validation pipeline and allowing invalid transactions to be re-executed. We can only allow this by restricting the results of the secondary execution step to manipulate the same set or a subset of the keys of the primary execution step. However, since the secondary step is intended to reproduce the work of the primary step based on an updated world state, this could be seen as a feature rather than a limitation. Nevertheless, our current proof of concept requires software developers to write two different smart contracts, one for each step. Future work could explore the possibility to merge this into a single smart contract to decrease the risk of software bugs and reduce the required maintenance effort.

From using of a different execution engine for the second than for the first execution step stems the biggest performance overhead in our proof of concept. To make the XOX model truly attractive for adaption in large scale blockchain systems, we need to integrate a more efficient execution engine. WebAssembly based execution engines appear to be a promising research direction.

Finally, in this work we only treat the symptom of invalid transaction, which is a systemic problem of the execute-order model. Orthogonal approaches to XOX that investigate the possibility of reducing the creation of invalid transactions during the primary execution step should be explored. Two possible solutions could be actionable programming guidelines for developers on how to model the key space of their application and intelligent routing of transaction proposals so that specific endorsers process all transactions that touch a certain portion of the total key space.

Chapter 5

TNG: A privacy-preserving blockchain protocol

Conceptually, blockchain systems make all stored information publicly available. However, especially in enterprise settings where collaborations between multiple companies exist, sensitive data should not leak outside of defined boundaries. Previous efforts to bring privacy and blockchains together either still leak partial information, are restricted in their functionality [94] or use costly and/or time-consuming mechanisms like hash time locked contracts or zk-SNARKs [90, 91]. Hence, we propose TNG, *the next generation* of blockchain privacy protocols that only relies on simple cryptographic primitives and targeted dissemination of information. We demonstrate its capabilities and show that despite strict privacy guarantees, the blockchain's trust and immutability guarantees still hold. The flexibility of this protocol allows us to introduce a simpler programming model for smart contract developers, which adds an additional layer of abstraction between the blockchain's internal data management and the application's business logic.

5.1 Motivation

In this section, we highlight two major drawbacks of Hyperledger Fabric's privacy features and programming model. These inadequacies inform the construction of our TNG framework.

5.1.1 Hyperledger Fabric's privacy features are inadequate

We start by defining the term *private data*. In the context of a collaborative and/or competitive enterprise setting, all data should be considered private by default, i.e. it is only visible to its originator(s). Data only becomes public if it must be shared with everyone. Once someone has access to a particular piece of data, this cannot be undone. Therefore, special care must be taken when defining access rights to data. Examples for private data are business contracts, movement of wares, sensitive customer information or financial figures.

As the most basic privacy feature, administrators can use Fabric's channels to completely separate private from public data. Unfortunately, since Fabric does not implement cross-channel communication, the usefulness of such an architecture is limited. Specifically, this approach only makes sense if private data is completely detached from public data. However, in this case, collaborators could simply create a channel for public data and keep their own private data in secondary non-blockchain data stores.

Starting in v1.2 [50], Hyperledger Fabric offers *private data collections*. When endorsers execute a chaincode, they usually create a public read and write set (RW set) as a response to a client's proposal. With private data collections, they can also read from or write to these collections that are stored in a local *transient store*. For each such collection, a policy is set up to manage which organizations can access it. However, this poses a problem for Fabric's validation step. Each peer, regardless of its organization, must be able to correctly validate transactions. It cannot do this if some keys are completely hidden from it. To solve this, endorsers that write to a private data collection add a hashed version of the private keys and values to the public RW set. These entries act as stand-ins for their pre-images. The version of a hashed key changes if and only if its pre-image was written to, so the validation of a hashed key is equivalent to validating its pre-image. This allows all peers to correctly validate every transaction regardless of private data.

This alone is not enough to enable private data in transactions. Endorsers must also keep a secondary write set for private keys in their transient store. Then, when they encounter a transaction with hashed keys during validation, they load the corresponding write set from the transient store and commit it to the private data collection if the transaction is valid. However, there might be peers in an organization that are authorized to access the private data collection that were not part of the endorsement of the transaction. Therefore, they do not have the private write set in their transient store. In that case, they need to ask other peers in one of the authorized organisations to share their private knowledge with them.

We illustrate this with a supply chain example. Assume there is a manufacturer that

buys raw materials from three competing suppliers. They all run a Fabric blockchain together, each acting as a separate organization that provides several peers. The administrators set up three private data collections. They install policies so that each of them can be accessed by one of the suppliers, but the manufacturer has access to all. Then, they install a chaincode that handles auctions on the ledger. This chaincode has three functions: `Start()`, `Bid()` and `Close()`. The manufacturer uses `Start()` and `Close()` to manage auctions, while the suppliers call `Bid()` to participate in existing auctions. Now, the manufacturer invokes the start of a new auction for 12 tons of copper and a specific expiry date. All of this data is publicly available and results in an RW set. This and all the following sets are represented in Figure 5.1. Subsequently, Supplier_1 and Supplier_2 send in a bid, while Supplier_3 cannot provide the requested material and refrains from this auction. Because the suppliers are competing, the auction implements private bidding. This means each supplier only writes one publicly available key to the ledger that announces their bids and gives it an ID for reference. Additionally, they write their quote to their private data collection, which can also be accessed by the manufacturer. In the end, after the auction has expired the chaincode to close it will be invoked by the manufacturer. The chaincode now reads all bid IDs from the public ledger to find the quotes which have been shared in the private data collections. Then, it finds the cheapest one and declares a winner. To ensure auditability by all participants, the chaincode also discloses which bids were considered. If a supplier gave out a correct bid which was not considered, they could verifiably dispute the result.

The existence of private data collections is of no importance for clients of the blockchain during transaction creation. They go through the same steps of endorser response collection and transaction submission as they would in the case of purely public execution. Based on the clients credentials in the membership service provider, they can either execute chaincode that interacts with private data or they receive an error response.

This protocol has several weaknesses. First, it is known which clients and endorsers are involved in the transaction, although the identity of clients can be obscured with Fabric's implementation of an identity mixer [18] that uses zero-knowledge proofs to verify to the endorsers that the client has sufficient authorization without giving away the actual credentials. Second, while all private keys and values are hashed, each unique key will still have a unique hash. This means that a curious observer can find out how many times a bid was placed by a competing organization, which might reveal their bidding strategy or other private information. Third, special care must be taken by the chaincode developer to ensure that private information cannot leak unexpectedly. For example, if the `Close()` function could be called by clients belonging to one of the suppliers before the auction has expired, then they could periodically simulate the outcome of the auction without

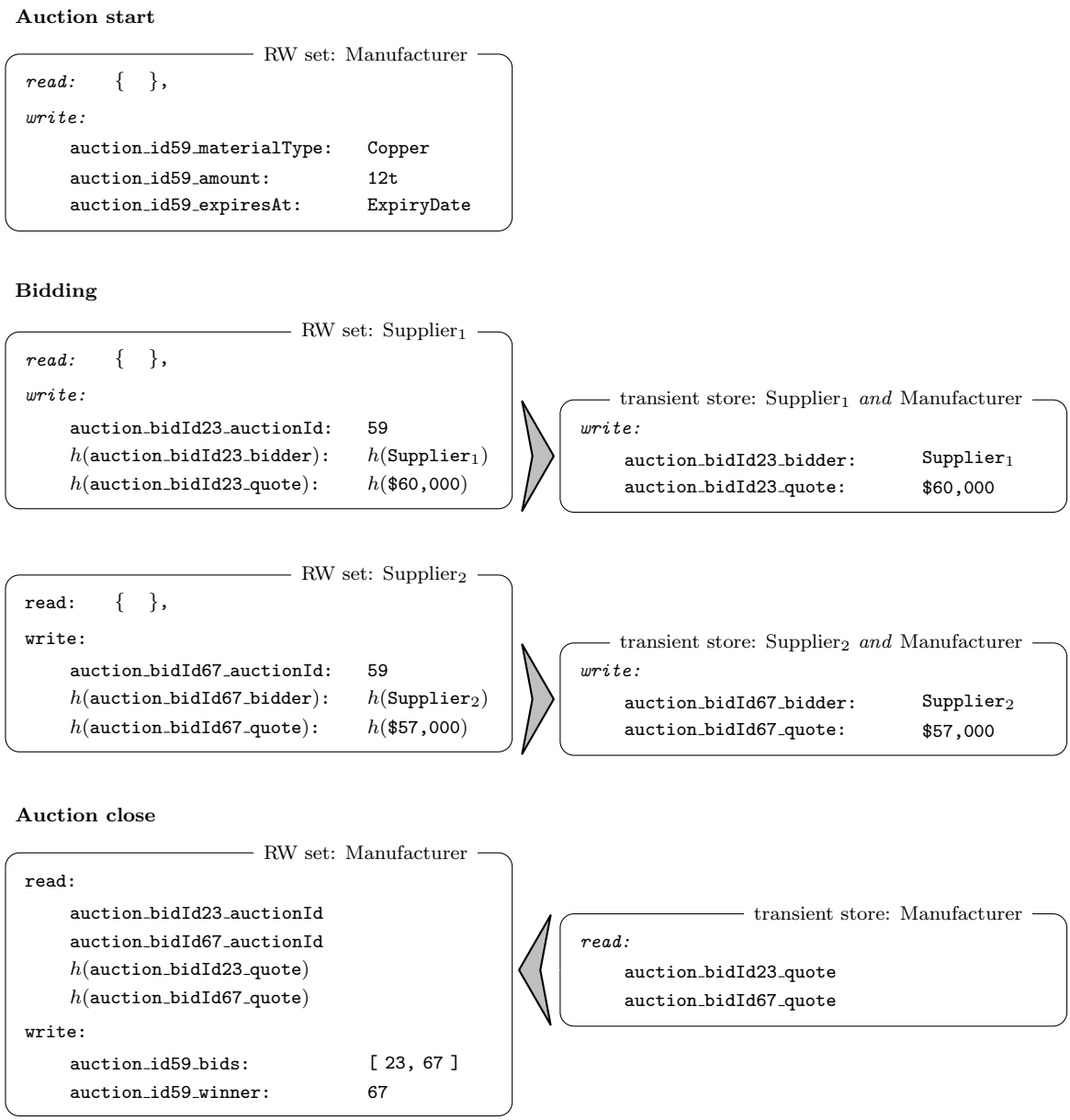


Figure 5.1: Illustration of RW sets for an auction chaincode including the interaction with the transient store. Supplier_1 and Supplier_2 enter bids with IDs 23 and 67 respectively to an auction with ID 59.

submitting the transaction to the ordering service. Then, whenever they saw a competitor winning the auction, they could submit a new bid. The last and biggest weakness of the protocol is that the proposal itself is still in cleartext. So, anyone in the network will know which chaincode was called and the corresponding parameters of the call. If, due to poor development practice, keys and values were parameters themselves they could be easily deduced. Therefore, private data collections leak a considerable amount of information that clients might want to keep secure. As we will show in Section 5.3, our TNG protocol completely prevents curious clients and peers from learning any information they are not privy to.

5.1.2 Hyperledger Fabric's programming model is inadequate

While the ledger is important as the backbone of the immutability guarantee of the blockchain and as an audit device, it might as well not exist as long as no faults occur. The actual business value of the blockchain lies in its world state. Chaincodes form the interface between the outside world and the blockchain world state. Fabric chaincode developers can use a defined API to manipulate the world state directly. First, we discuss this API and its shortcomings. Then we examine the weaknesses of Fabric's chaincode model due to its lack of composability.

Fabric's world state API

The essential API to interact with the world state consists of the functions `PutState`, `GetState`, `DelState` to create or update, query and delete states respectively; their counterparts for private data are `PutPrivateData`, `GetPrivateData` and `DelPrivateData`. Furthermore, some variants of range queries can be performed. However, most range queries are not safe to use in transactions modifying the world state, because it cannot be guaranteed that the range that was queried still holds the same items during commitment as during chaincode simulation.

This API forms a very thin layer of abstraction over Fabric's internal key value store. Whereas many conventional databases, especially databases implementing SQL, have an additional database abstraction layer (DBAL), Fabric only has direct interactions with key-value pairs. As a consequence, chaincodes are tightly coupled to the database structure. This puts the burden of creating a translation between business objects and key-value pairs onto chaincode developers. Additionally, it makes any evolution of Fabric's internal data storage very difficult without breaking most if not all existing chaincodes as we will demonstrate shortly.

Chaincode

A chaincode is a collection of functions that interact with the world state and which all share the same namespace. For example, the chaincode that describes the interaction with a bank account could consist of functions to `open` a new account, `close` an existing one, `query` an account with a specific *id* and `transfer` money from one account to another. All of these domain specific actions must be translated into the manipulation of key-value pairs to make use of the previously described API. Each key touched by a chaincode is stored internally with the same chaincode namespace prefix.

To illustrate this, consider the chaincode for a bank account that is installed in the namespace `ledgerBankAccounts`. Alice would like to open a bank account with the fictional *LedgerBank* so she can pay for the fictional movie streaming platform *StreamMore*. First, the chaincode function `open(`*`id:`*`Alice,` *`initialValue:`*`100)` is called to open the account for her. To store her new account in the world state, it could be translated to the internal key-value representation {`ledgerBankAccounts_accountAlice: 100`}. With the account in place, she wants to pay StreamMore to get access to their platform. To this end, she could make use of the `transfer` function of LedgerBanks chaincode. However, StreamMore does not have an account with LedgerBank. This is a problem, because chaincodes cannot modify keys outside of their own namespace. This means `transfer` cannot easily reach the StreamMore account. The only way this can be achieved is by calling the StreamMore chaincode directly from inside the LedgerBank chaincode.

We assume that LedgerBank has implemented such a bridge because it was a frequent request. To get access to the streaming platform, Alice must call StreamMore's `getAccess` function. However, after the payment went through, the call to `getAccess` might be denied. This is possible, because payment and service access were not linked by an atomic transaction. Such a scenario must always be taken into account, when developing for a blockchain[1]. Indeed, if the participants could rely on trust, then a blockchain is not needed. The only way to make this exchange atomic in Fabric, that is, to couple a payment to StreamMore to access to their platform, is to use the same chaincode-to-chaincode mechanism that enabled the payment.

There are two possible implementations for this. The `transfer` function of LedgerBank could take in additional parameters and call `getAccess` in StreamMore's chaincode. However, StreamMore's platform is not the only one that relies on payments for services. So LedgerBank would need to implement specific logic in its transfer function for each of these services. This is not feasible, because it breaks the principle of separation of

[1]This is the reason why hash timelocked transactions were developed in the context of Bitcoin

concerns [30] and the bank's chaincode would need to be updated for every new service. Moreover, Alice would somehow need to prove to StreamMore's `getAccess` function that it was called from inside LedgerBank's `transfer` function, which transferred the correct sum to the correct account. On the other hand, the function calls could be reversed so that `getAccess` calls `transfer`. This would solve the separation of concerns to some degree, because StreamMore should have knowledge of all its supported payment services. Therefore, this dependency appears more reasonable than the other way around. Yet, this would mean channeling privacy-critical data related to Alice's interaction with her bank directly through StreamMore's API. Thus, both approaches have significant drawbacks. In addition to these privacy concerns, Fabric imposes the restriction that for chaincode-to-chaincode calls all involved chaincodes must be installed on the same endorser. A cross-peer chaincode-to-chaincode invocation is not allowed.

Creating a blockchain based payment ecosystem would mean that there must be a link between each payment provider and each service provider. This means for n payment providers and m service providers there must be $n \cdot m$ implemented links. Moreover, each peer would need to have hundreds if not thousands of chaincodes installed. Each of these chaincodes creates its own Docker container upon instantiation. This is already problematic from the point of view of the Docker engine [71]. What is more, all chaincode execution is governed by endorsement policies. This means that for each transaction, multiple endorsers will have to execute a chain of chaincodes to create their response for the client, causing a significant amount of replicated computation.

In conclusion, Fabric's current chaincode model ties business logic tightly to its internal data structures and makes the creation of large ecosystems infeasible, because it leaks private information, puts too much burden on developers and has excessive computational overhead. We will introduce a new programming model in Section 5.2.3 which, in tandem with the TNG privacy protocol, eliminates all these problems.

5.2 Architecture

For easier understanding, we start by explaining TNG in terms of Hyperledger Fabric. We present a proof-of-concept implementation based on Fabric in Section 5.5. After discussing the basic protocol, we will expand TNG in Section 5.6 to work with any permissioned blockchain framework that has a distinction between replicating nodes and ordering nodes. In the following, we will assume that communication is point-to-point authenticated, e.g., using TLS [98], and communication in the network is partially synchronous [32]. More

specifically, there is an upper bound on latency $\Delta < \infty$ for every message sent between two correct network nodes.

5.2.1 Clients

Instead of using the default endorsement and chaincode protocol, clients call the new TNG protocol to interact with the blockchain. More specifically, transaction creation becomes a collaborative process, and multiple clients can be involved in the creation of a transaction. Note that each client may be Byzantine faulty, that is display arbitrary malicious behaviour. In particular, some or even all clients involved in a transaction might collude against the rest of the network.

5.2.2 Nodes and shards

To make it easier to generalize the protocol later, we will call Fabric peers *nodes* in the context of TNG. While sharding is commonly used to enhance performance, in this work we use it to increase the privacy capabilities of the blockchain system. In TNG, every *node* in the network belongs to at least one *privacy shard*. These shards, like organizations in Fabric, form trust boundaries. Every node in the network stores the full blockchain ledger, but the world state is completely sharded into distinct non-overlapping partitions. This means that each node shares the same partial world state with all other nodes in the same shard, but has no key space overlap with nodes from other shards. Only a union that includes all shards represents the entire world state. Note that a node can belong to multiple shards simultaneously. In this case, its local state database will contain the union of partial world states of all shards it belongs to. We assume that all nodes and their shard memberships are known to all other nodes in the network, just as peers and organizations are known in Fabric.

In our framework, nodes can exhibit Byzantine behaviour with the restriction that they must not leak private data to unauthorized parties. In practice, this means TNG can handle unexpected hardware and software bugs, but not intrusive hacking. The only ways to completely curb the risk of information leakage are either completely isolated disjoint data silos or heavy use of zero-knowledge proofs, both of which are unfit for a high performance privacy-preserving blockchain. Furthermore, we believe that in an enterprise setting, uncontrolled dissemination of private information by trusted shards is less of a concern than untrusted shards learning more than they should. The introduction of Trusted

Execution Environments as discussed in Section 5.7 and Section 5.8.5 could be a way to extend our protocol to handle malicious nodes trying to leak private data.

To deal with incorrect chaincode execution by a node, we adopt Fabric's endorsement policies. These guarantee that the result of the chaincode execution from multiple nodes must agree. For f faulty nodes, the policy must require to receive $f + 1$ equal results. Then, Byzantine nodes either have to agree with the correct result of a well-behaved node or are excluded from the chaincode execution. However, for the benefit of data recovery, we assume that an entire shard never fails. Recall that if all of Fabric's organizations that share a private data collection fail, then that collection cannot be recovered. Similarly, the partial world state of a shard is lost if it fails completely, because no other shard replicates it. This makes a shard the custodian of the data its nodes store in their state databases. Therefore, a shard has the responsibility to keep the private data of its clients safe. While it is impossible to prevent a malicious node from leaking information that it is authorized to access, TNG prevents any unauthorized malicious agents, be it nodes or clients, from learning any private information.

In contrast to siloed private data stores in Fabric channels, TNG allows atomic cross-shard transactions. To allow this we slightly weaken blockchain's usual trust assumptions. This is inevitable as soon as private data comes into play, because clients have to trust any host of their data to not disseminate it indiscriminately. Since TNG's shards act as data hosts, a client must trust one or more shards to keep their data private[2]. If two clients want to interact on the ledger, then they must agree on which subset of shards to trust with their private data. If the clients are concerned that nodes might disseminate their private information to unauthorized parties they can store their data in encrypted form instead. In this case, they use TNG's smart contract equivalent to retrieve the encrypted data, manipulate it off-chain and, with a second call, store the encrypted new value back on the blockchain. This trades increased privacy for a loss of replicated computation.

5.2.3 Smart contract execution

As discussed in Section 5.1.2, Fabric's chaincode model is unsuited for a collaborative privacy-preserving blockchain. We therefore replace it with *smart assets* as the basic building blocks. Before we describe this new model, we introduce the notion of *stakeholders*.

[2]If aggregation in a single place should be a concern, then a client can spread its private data across many shards to make targeted profiling by nodes more difficult

⋮	
key: ledgerBank_account_id1234_owner	*value:* Alice
key: ledgerBank_account_id1234_balance	*value:* 100
⋮	

Figure 5.2: A simple bank account mapped into a key-value store structure

Stakeholders

According to Section 5.2.2, shards store distinct partitions of the world state. To make these partitions private, access to the data must be restricted. We do this by assigning owners to all data. Then, we can implement rules based on this ownership to restrict or grant access. The obvious choice for data owners of private data are the clients that put the data on the shard in the first place. But this would allow the shard to connect its data to specific clients. TNG prevents this by adding the additional abstraction of *stakeholders*. Stakeholders are entities that stand to gain or lose something from the modification of specific data, i.e. they have a stake in it. From this point onward, we make the distinction between stakeholders, the originators of transactions and owners of private data, and clients, the physical manifestation of the stakeholder's interaction with the blockchain. In particular, while clients must have a known network identity, stakeholders are anonymous. As we will discuss in Section 5.3.2, this allows stakeholders to use clients as proxies, so that the blockchain shards never learn who the actual transaction originator was. In the following we explain how we can use anonymous stakeholders to restrict data access.

Smart Assets

To make use of stakeholders and proof of ownership for data access, all data must have ownership information attached to it. With Fabric's key-based data structure derived from its internal data store, it is difficult to prevent unauthorized access to individual keys. Based on our definition of assets in Chapter 2, we introduce the abstraction of a *smart asset*, inspired by object-oriented programming paradigms. We start the construction of this new model by encapsulating all properties of an asset that can be manipulated by a Fabric chaincode into a single object. Consider the example of a bank account. As a minimal implementation, it needs to have the properties *owner*, and *balance*. Instead of invoking a chaincode which manipulates keys in a key-value store as shown in Figure 5.2,

stakeholders create transactions by interacting directly with the smart asset `BankAccount`. We call this object a smart asset, because it completely governs the access to its own data. To make this possible, we require every smart asset to implement at least two functions:

`GetID()` Every smart asset has a unique identifier. The ID is used to find it in the data store.

`GetStakeholders()` Without the notion of ownership, we cannot implement any access control. This function returns the public key(s) of the owner(s) of the smart asset. Note that multiple stakeholders can own a particular asset. For example a joint bank account can be owned by two stakeholders. These public keys remain the only identifier of the involved stakeholders in the TNG protocol. For maximum anonymity, a stakeholder can generate a new private-public key pair for every asset they interact with.

In TNG, instead of installing chaincode, administrators install smart asset definitions on TNG's shards. These define the API of the smart asset. Besides `GetID()` and `GetStakeholders()`, the smart asset `BankAccount` could contain `Open(), Withdraw()` and `Deposit()` methods to manipulate private data. While this looks similar to how a chaincode would be defined, both the idea and implementation of smart assets are radically different. Note how every method only manipulates the asset itself. There is no `Transfer()`, because this would require one bank account asset to manipulate another. We will show in Section 5.2.4 how we can still achieve the same functionality. Furthermore, the asset's API is the only way to manipulate its state, the state itself is private. This allows us to provide method-level fine-grained access control.

We make the observation that the set of stakeholders that has to agree on an action changes depending on the action. If we take the example of a joint bank account that is co-owned by Alice and Bob, then they would both need to agree to open or close the account. But each of them should be able to individually withdraw or deposit money. Therefore, we require each call to an asset's API method to be confirmed by a set of stakeholders. To this end, we need to put two more pieces into place. First, instead of invoking the asset's method with a list of parameters, we pass it a *data transfer object* (DTO) that encapsulates all necessary data. The smart asset's function implementation can then extract data from the DTO as needed. However, the DTO also implements a `GetStakeholders()` interface that returns the stakeholders of the state transition that is invoked by the call to the corresponding asset method. During such a call we have now access to the set of stakeholders passed in by the DTO and the set of stakeholders that

owns the smart asset. Second, we need a rule that either allows an action or prevents it. Analogous to the endorser policies we call this set of rules the *stakeholder policies*. When the asset definition is installed, the administrators also need to setup a stakeholder policy for every function in the API individually. With this, the TNG protocol can interject itself into a call to a smart asset, call `GetStakeholders` on both the DTO and the smart asset and pass the two sets to the corresponding stakeholder policy. If the policy is successfully validated, the function call is granted, otherwise the system denies it. This mechanism is completely independent of the implementation of either the smart asset or the DTO, as long as both implement the `GetStakeholders` interface. In the case of the joint bank account, a system administrator would install a "*must-match-all*" policy for `Open` and `Close`, but a "*must-match-one*" policy for `Withdrawal` and `Deposit`. Because the stakeholder policy uses public keys to identify stakeholders, any malicious actor could potentially impersonate a stakeholder. To prevent this the stakeholder policy also checks corresponding signatures which can only be generated with the stakeholders' private keys. We will describe this mechanism in more detail in Section 5.3.2.

The stakeholders of an asset can change over time, if one of its functions allows that. For example, we can emulate Bitcoins UTXO model by creating a *tngCoin* smart asset. We add the function `Give()` to its API. This function takes in a DTO that encapsulates the new owner of the coin. Furthermore, we define a stakeholder policy that expects to get two stakeholders from the DTO, one of which must match the current stakeholder of the coin. If the policy is successful, the `Give()` function replaces the old owner with the new one.

This illustrates that stakeholders not only govern access to information, but also prevent double spending as long as it is guaranteed that the ledger cannot fork. Whenever a stakeholder transfers ownership of an asset to a new stakeholder, the asset will not accept the old stakeholder's signature anymore. If a malicious stakeholder tries to create two transfers for the same asset they will eventually appear totally ordered in the global ledger. Then, the first transaction changes the stakeholder, so when the second transaction is validated it violates the stakeholder policy of the asset and is discarded.

Once an asset has been created or modified, it needs to be stored in the state database. Instead of spreading the asset properties across multiple keys like we showed in Figure 5.2 we simply serialize assets into bit streams, store them by their IDs and deserialize them when needed. This circumvents the risk that individual keys are being modified and transition the asset into an inconsistent state. This way, a key-value store like Fabric's LevelDB state database essentially becomes a *BLOB*[3] store. Because this storage process is now

[3]for Binary Large OBject

identical for each asset, it can easily be automated and incorporated into the blockchain's data layer. This completely removes the need for asset developers to know about the internal data structure of Fabric. They only need to create serializable objects which implement `GetID` and `GetStakeholders`.

Note that smart assets only control their own behaviour. All interaction between multiple assets is delegated to scripts that invoke methods on smart assets. Thus, smart assets form the foundation on top of which all other business logic can be built. In particular, we will show how this model allows for easy composability in Section 5.3.6. This separation of low level assets and higher level interactive business logic has the added benefit that development now can be handled by different domain experts, neither of which has to be a blockchain expert. Consequently, not only do application developers not need to care that they interact with a blockchain, even developers of asset definitions can focus on the description of the asset's properties and functions and let the blockchain handle the storage.

5.2.4 Transaction structure

A transaction in Fabric contains exactly one proposal for a chaincode execution with all its input parameters, as well as the execution result and possibly multiple endorser signatures. This is also true for transactions with private data collections except that some keys and values in the transactions RW set are hashed. This offers a curious observer plenty of information. In contrast, with TNG, a transaction contains only hashes and signatures and no cleartext. Clients still create cleartext *proposals*, messages that contain all necessary data to trigger endorser execution, and send it to endorsers. In response, the endorsers execute a smart asset function, but instead of returning the result, they only send a hash back. The details of this procedure will be explained in Section 5.3.2. Importantly, each such hash fulfills the same function as the hash stand-ins in the case of private data collections: Once a node recognizes a hash in a transaction during validation, it will lookup the corresponding state transition in its internal transient store and commit it if the transaction is valid.

Each hash corresponds to one call to a smart asset and a transaction can contain an arbitrary number of them. Notably, each call can involve a different shard. As a consequence, only the endorsers of the shard that was called know the pre-image of the hash. While they can share that information with other nodes in the same shard, it would be a break of the protocol to disseminate it outside of the shard's bounds. This means that for a transaction that involves multiple shards, there is no single entity that knows the full

pre-image of a transaction. Furthermore, if a shard comprises three hashes and only one of them was produced by shard A, then TNG does not allow shard A to learn who produced the other two. Yet, transactions are commonly associated with atomic execution. In the following we explain how TNG still achieves atomicity.

We compare two cases. First, two regular transactions are submitted to Fabric, each modifying independent keys. We have shown in Chapter 4 that we can validate them simultaneously and their result is independent of each other. Consequently, we can either discard or commit their results based on their individual validity. However, compare this to a transaction with chaincode-to-chaincode linking. Either the result of both chaincodes is valid or the whole transaction must be discarded. We note that atomicity is tightly linked to a dependency between different parts of an execution.

Because in TNG the stakeholders are the only ones with full knowledge of the goal of the transaction they must be the ones to create dependencies between the separate smart asset executions. To this end, they add *dependency sets* to their proposal. When a proposal is sent to a shard, then the dependency set of this proposal contains the identifiers of all shards this proposal is dependent on. Based on these identifiers the shard can then later look up all the nodes it needs to contact during the validation step. This step is described in detail in Section 5.3.2. It is unavoidable to disclose dependency information to the shards, otherwise the creation of atomic cross-shard transactions becomes impossible. To illustrate this, we return to the LedgerBank and StreamMore example. Alice creates a transaction that comprises proposals to execute `Withdraw()` on her own bank account and `Deposit()` on StreamMore's account. Additionally, she proposes to create a new AccessToken smart asset that gives her access to the streaming platform. Assuming all three assets are stored on different shards A, B and C, then the dependency set she adds to the proposal for shard A contains B and C. Similarly, the dependency set for shard B contains A and C and the one for shard C contains A and B. This way, all parts of the transaction are completely dependent. If one of these were missing, the shard would have no knowledge that its part of the transaction is linked to the others. Our atomic commit protocol described in Section 5.4 ensures that all parts commit atomically.

However, because there is no all-knowing auditor, stakeholders can also create transactions with parts that are not fully connected by their dependency sets. As long as all parts are at least sparsely connected, our protocol still guarantees atomicity. But once the dependencies form disconnected subsets atomicity is only guaranteed among each subset. For example, if Alice forgets to add a dependency set to the proposal to call `Deposit()`, then `Withdrawal()` and `CreateAccessToken()` still commit atomically, but it is not guaranteed that `Deposit()` reaches the same result. Still, each smart asset execution on each shard is executed (or discarded) correctly and none of the partial world states becomes

inconsistent no matter the outcome. This is because dependencies between different parts of a transactions do not symbolize consistency, but intent. A bank account will never arrive in an inconsistent state because another shard fails to modify the tngCoin that was instantiated there.

This is analogous to submitting two conventional transactions and expecting the system to commit both. Therefore, it is up to the stakeholders to properly declare their transactions. We could redefine transaction to mean any dependent subset of transaction hashes, but in order to stay congruent with blockchain jargon we keep the current meaning and accept that transactions as a whole do not need to be committed atomically.

Due to its stakeholder policies, TNG also prevents malicious stakeholders from using this to take advantage of uninvolved stakeholders. A smart asset can only be modified by their stakeholders, so any unwanted change that results from malformed dependency sets can only involve assets of the stakeholders that created the transaction.

5.2.5 Ordering service

The ordering service is responsible for creating the global order of transactions. As such, the ordering service acts as an honest but curious observer. It creates a total order of transactions submitted by clients, cuts this order into blocks and disseminates the blocks to all privacy shards in the network. We show in Section 5.3.2 that, by construction, the ordering service does not know which shards are involved in a single transaction. Therefore, it is impossible to disseminate transactions only to involved shards. Furthermore, we also demonstrate in Section 5.3.5 that the global ordering service cannot be split into multiple independent parts.

5.3 Simplified TNG

Before we extend the protocol to a fully decentralized scenario, we explain a simplified version of TNG, which replaces some actors of the network with black boxes to facilitate understanding of our protocol. In particular, we abstract shards and the ordering service as single nodes.

5.3.1 Guarantees

We want to guarantee the following properties:

(I) A well-formed transaction submitted by a well-behaved client will eventually appear in the ledger. (Section 5.3.2 Ordering)

(II) If a well-formed transaction has been submitted to the ledger, then any atomic part of the transaction will eventually be unanimously committed or unanimously discarded by all involved privacy shards. (Section 5.3.2 Validation & Commitment)

(III) Any malformed part of a transaction will never lead to an inconsistency and will not break atomicity of dependent transition commitments across all involved shards. (Section 5.3.2 Validation & Commitment)

(IV) The only ways for a network participant to learn about the state of an asset in a privacy shard are:

  (i) It is a part of the shard

  (ii) The shard itself leaks data, i.e. breaks our threat model assumption

  (iii) A client leaks information about an asset of which that client is a stakeholder.

  (Section 5.3.4)

5.3.2 Transaction flow

In the following, we present the simplified TNG protocol in detail. For easier understanding, we explain the general message flow step-by-step with the concrete example of an IOU use case. an overview of the single steps is presented in Table 5.1

IOU use case scenario

One stakeholder, Bob, would like to borrow 100 coins from another stakeholder, Alice. For this purpose, they will create an IOU ("I owe you") smart asset denoting the borrowed sum on privacy shard s_{iou}, which has the definition for this asset installed. In exchange, Alice will transfer an existing asset worth 100 coins that is stored in privacy shard s_{coin} to Bob. These two operations must be executed atomically similar to a payment channel in a permissionless setting. We assume both Alice and Bob act well-behaved and each use their own client to interact with the network.

Protocol overview		
Repeat	Negotiation	Clients agree on a common proposal
	Client Proposal	One client sends the signed proposal to a shard
	Shard Response	The shard stores the pre-image and responds with a signed hash
Transaction creation		All signed hash responses are collected and all stakeholders sign the Merkle root
Ordering		Transactions are sorted into blocks and disseminated to all shards
Validation & Commitment		Shards check transactions for known hashes, validate the result of the pre-image and agree with any other dependent shards on discarding or committing the transaction

Table 5.1: Summary of the transaction flow protocol

Negotiation phase

Alice and Bob negotiate the content of the DTO for the `CreateIOU` call. In particular, they need to ensure they both add their public keys to mark them as stakeholders and agree on the correct value which the IOU should denote after creation. This negotiation happens completely between the involved clients without the involvement of any shard or ordering service. The IOU asset definition offers the `CreateIOU` function to create new IOU assets. The stakeholders of an IOU asset are the involved borrower and lender. These are set by the DTO that works as the input to `CreateIOU`. Therefore, Alice and Bob need to ensure they fill the corresponding properties of the DTO with their corresponding stakeholder keys. To this end they can either each create a completely new private-public key pair to reuse existing ones, depending on their privacy preference. If they reuse an existing public key as the stakeholder identifier, then the shard they communicate with is able to connect different assets to them by matching their keys. The creation of the DTO happens completely off-chain in an interaction between Alice and Bob.

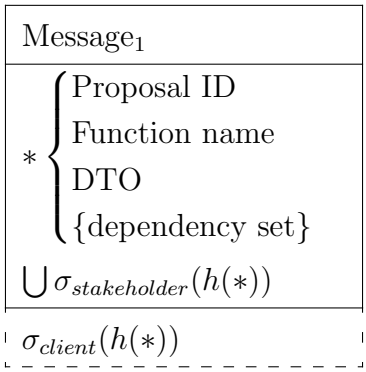


Figure 5.3: Client proposal

Preparing a Proposal

Once both Bob and Alice are satisfied with the DTO for the IOU creation, they prepare the proposal to a shard we call s_{iou}. This proposal consists of

- an arbitrary proposal ID,
- the uniquely specified name of the function call, in this case `CreateIOU`,
- the corresponding DTO,
- a set of shards this proposal is dependent on.

The proposal ID ensures that a shard receiving two otherwise identical proposals can differentiate between a malicious/accidental replay of the same proposal and a true second proposal that just coincidentally has the same parameters. The dependency set denotes which shards need to collaborate to make this transaction atomic. By adding $\{s_{coin}\}$ as the dependency set to the proposal, Alice and Bob can tell s_{iou} that it needs to coordinate with s_{coin}, which has to validate another part of the transaction. In this case, s_{iou} will only commit the changes if s_{coin} also commits its changes. We will discuss validation later in this section.

When all the proposal parameters are set, then the stakeholders need to sign it. This is done by signing the Merkle root hash of the proposal ID, the function name, the DTO and the dependency set with the private key that corresponds to their stakeholder public key (Figure 5.3). In the context of message signatures, we use the Merkle tree structure

as a deterministic procedure to generate and validate the message hash. Furthermore it potentially allows clients to partially reveal single fields of the message for auditing purposes. This can be done by revealing the field in question and providing sub-tree hashes for an auditor to reconstruct the Merkle root like we depicted in Chapter 2, Figure 2.3. Neither the Merkle tree nor its root hash are actually sent with the message because the receiving shard can reconstruct it and checks if all stakeholders have signed the message. For `CreateIOU`, the installed stakeholder policy on s_{iou} validates that both the lender and borrower sign the proposal, so both Alice and Bob add their signatures. Given these signatures neither Alice nor Bob can create a valid proposal that has not been seen by the other.

Finally, Bob, who is sending the proposal to the shard, adds a signature associated with his client's role in the network to the message (Figure 5.3) to verify that he is allowed to make proposals to a shard. An example for such a role would be *WRITER*, someone who is allowed to create proposals and transactions. This last signature is not considered part of the message and will be discarded by the shard after verification, so that no information about the sender of the proposal can leak to the rest of the network. By making the distinction between client signatures and stakeholder signatures we separate different concerns. Client signatures govern access to the blockchain network, stakeholder signatures govern access to the private data on the shards. If a client becomes malicious then the network administrators can retract its privileges or even eliminate it from the network. This does not lock out the stakeholders from their assets. In turn, a client who is ignorant of the stakeholders' private keys cannot take over their assets. Lastly, stakeholders can encrypt the message content and use a proxy client like a broker to send proposals to shards and the client will not learn the content of the proposal.

Because of the proposal ID, each signature is tied to a specific proposal. It cannot be reused by a malicious client without trying to replay the whole proposal, which can be dealt with by duplicate prevention on the shard.

Response

When the shard s_{iou} receives the proposal from Bob, it first checks the proposal for duplication, then it unpacks the payload and validates the signatures against the stakeholder policy of `CreateIOU`, using the `GetStakeholders()` interface of the DTO to receive the necessary public keys. If this validation is unsuccessful, the shard simply responds with an error message. Otherwise, the shard now simulates `CreateIOU` just like a regular Fabric endorser would. It signs the Merkle root of the proposal message (without the client's signature) and the simulation result. The signed proposal plus simulation result correspond to

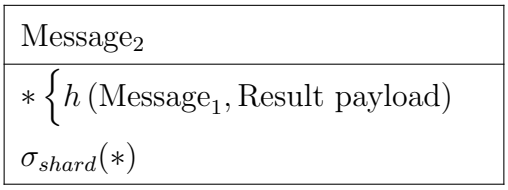


Figure 5.4: Shard response

Fabric's endorser response. But, instead of sending this back to the client, the node shard now hashes this pre-image and stores it in its local transient store with the hash as its key. Then, the shard only returns the hash and its signature back to the client (Figure 5.4). Note that in this section, we treat the shard as a single black-box. We generalize this behaviour to a decentralized setting and even to different execution models in Section 5.6.

We point out that neither the client nor the stakeholders need to learn of the internal state transitions that result from their proposal. They only need to be concerned that their proposal is committed if their transaction is valid. To this end, the response hash will act as an anchor for the locally stored pre-image similarly to how hashed keys are treated by Fabric's private data collections. This further decouples the underlying blockchain from the application.

Multiple proposals per transaction

These three steps must be repeated for all proposals that make up the complete transaction. Therefore, Alice and Bob also have to agree on which asset Alice will transfer to Bob. They create a proposal `TransferCoin` to be send to s_{coin}, where that particular asset is held. They add the dependency set $\{s_{iou}\}$ to link it to the previous proposal to s_{iou}. Note that even though the shards learn about the other one's involvement, they do not learn anything about the other's proposal content.

The `TransferCoin` object must include the asset's ID, so s_{coin} can load the asset from its database during the simulation. They add Alice as the current owner of the coin and Bob as the new owner, because the stakeholder policy for the coin is given by:

> *One signature must match the current stakeholder of the asset, the other signature must match the new stakeholder of the transfer DTO.*

In general, multiple proposals to different shards can be handled concurrently, because their simulation is not linked. Only during validation do shards contact the other shards denoted in the dependency sets.

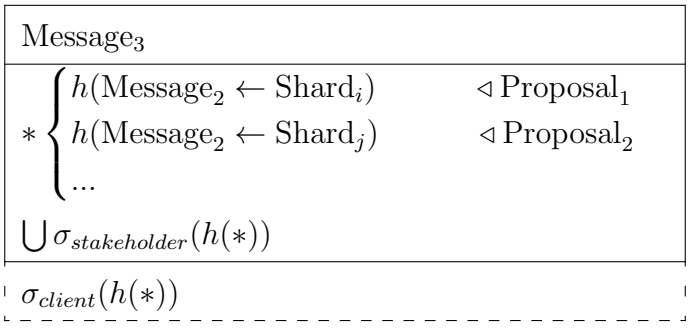


Figure 5.5: Transaction

Transaction creation

After Alice and Bob received the responses from shards s_{iou} and s_{coin}, they collect and hash them before both sign the Merkle root of all these hashed responses with the same private keys they used for the individual proposals. Just as with single proposals before, Alice attaches a signature associated with her client's network role (Figure 5.5). They now have created a TNG transaction.

A transaction in TNG consists of the hashes of all shard responses that are involved in that atomic exchange and the signatures of all involved stakeholders. Assuming the stakeholders have set the dependency sets correctly, the protocol will guarantee that every part of the transaction commits atomically. With their signatures the stakeholders signal that they have seen the composed transaction and agree to submit it. This will be validated by the shards later on. From the perspective of a shard, the only way to identify a stakeholder is by their public key. They count each public key as an individual stakeholder, even though they might belong to the same person. Therefore, if the stakeholders used different keys to sign individual proposals, then they need to sign the transaction with all of these keys. Before submitting the transaction to the ordering service, the client signs it without the need for any knowledge of the content of the pre-images that are anchored by their hashes. Once again, this signature is not used to validate the transaction content, but only to manage network access.

Transactions can implicitly involve multiple hashed responses from the same shard. We will discuss how to remove any arising ambiguity once we have explained the validation step.

Message_4
$h\,(\text{Message}_3)$

Figure 5.6: Validation request

Ordering

When the ordering service receives a new transaction, it verifies the client's role, then discards that signature and puts the transaction into a new block, which is then disseminated to the whole network.

Because the ordering service only sees hashes and signatures associated with unknown public keys, it cannot learn anything about the identity of the stakeholders or even which privacy shards are involved. However, it can learn the number of involved shards and stakeholders. If this leakage is an issue, clients can add any number of bogus hashes and fake signatures to the transaction in the previous step to obscure the true number of participants.

Because in the simplified model the ordering service is a black box with the intended properties, it fulfills guarantee (I) trivially.

Validation & Commitment

Both shards s_{iou} and s_{coin} scan every transaction in each new block they receive. They compare each Merkle leaf of a transaction with the hashes of pre-images in their transient store. At some point s_{iou} will recognize the hash for the response to the `CreateIOU` proposal. To check that the known hash is part of a valid transaction, s_{iou} verifies that all stakeholders of the pre-image have also signed the full transaction. If that were not the case, then the transaction could have been created without the knowledge of one of the proposal's stakeholders.

Because the simulated result of the proposal could be stale at this point due to an earlier transaction modifying the IOU asset, s_{iou} needs to verify that the result is still valid. This is identical to Fabric's current validation step.

If the result is invalid, it discards the transaction and moves on. If the result is valid, it asks s_{coin} if its part of the transaction was valid as well, because it knows of s_{coin} due to the dependency set inside the proposal. To identify which transaction s_{iou} needs an answer for, it sends the Merkle root of the transaction in question to s_{coin} (Figure 5.6). Note that

for transactions larger than two hashes s_{iou} does not know which other hash belongs to which shard, so it must use the hash of the whole transaction.

In the meantime, s_{coin} has gone through the same validation step, so each will respond to the other shard's inquiry with either a success or failure message. If a shard receives a failure message it discards the local result no matter its own validation result to achieve atomicity. If both results are valid, the result will be committed to the local ledger state. This ensures that either both the IOU is created and the coin is transferred or neither is, thereby adhering to guarantee (II). This commit protocol gets more complicated in a scenario with more than two shards. To better reason about the properties of this validation consensus mechanism, we encapsulate it into an isolated atomic commit protocol. We will discuss this protocol in detail in the Section 5.4 and prove guarantee (II) more rigorously.

Because the hashes that are composed into a transaction are produced by shards themselves, they can ignore all hashes they do not recognize. Either these hashes where created by other shards or are invalid hashes added to the transaction to obfuscate the real number.

During the validation step, we deal with an ambiguous corner case. A transaction could contain multiple proposals to the same shard. To break this ambiguity, we impose the following rule: Each shard must process the validation of computed results in the same order as their associated hashes appear in the transaction. This order is fixed by the signed Merkle root of the transaction, so the outcome is deterministic as long as the outcome of every single validation is deterministic. This eliminates inconsistencies among nodes of the same shard.

Next we need to address this in the context of cross-shard validation. The only shared knowledge between shards is the ledger. Therefore, all the information a shard can gain is from the pre-images it has stored in its transient store and their association with a specific transaction on the ledger. In particular, it cannot learn anything about the proposals that are associated with unknown hashes in the same transaction and which other shards they belong to. So, when a shard encounters a dependency on another shard, it cannot pinpoint the exact proposal it is dependent on. Instead, it can only ask the other shard to give a validation result for the entire transaction. The response to that inquiry should also not leak any information about the other shard's internal state. Therefore it must always be a single value, no matter how many proposals the other shard actually validated. The only way to neither break the internal consistency of the shard's partial world state nor the atomicity of the transaction is to aggregate all the shard's validation results for a single transaction in an all-or-nothing manner. This way, a single invalid proposal leads the shard to discard the whole transaction. The validation response messages includes the initial inquiry, so the receiving shard can associate the response with the correct transaction (Figure 5.7).

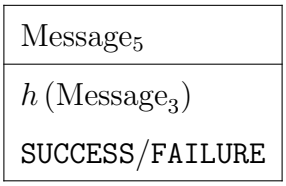
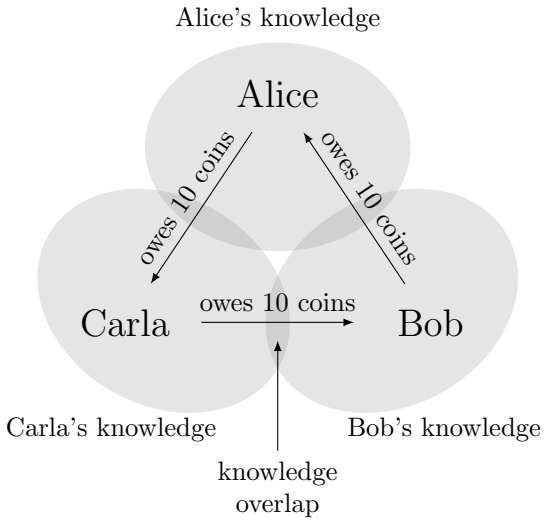


Figure 5.7: Validation response

We can introduce a dependency analyzer as described in Chapter 4 to allow multiple independent transactions to enter cross-shard validation concurrently. However, even with a fully sequential validation, it is guaranteed that all shards make progress eventually under the assumption that there are no permanent network partitions. By the time a transaction is submitted to the ordering service, all involved shards have seen their own associated pre-images. Otherwise, the shard would not have been able to sign the response to the proposal. If a shard does not recognize a hash in a transaction, this hash either belongs to a different shard or it is an invalid hash. So, it can be ignored in either case, hence guaranteeing (III). Therefore, when shards start the atomic commit protocol for a specific transaction, each involved shard can definitely decide its own start value. The delay for commitment is determined by the latest shard to receive the block and the latency of the atomic commit protocol. By assumption all shards are well-behaved, so all will eventually decide.

5.3.3 Transaction atomicity

After the detailed description of the TNG transaction flow we return to the issue of atomicity with an example. Take a netting scenario between the three stakeholders Alice, Bob and Carla as illustrated in Figure 5.8. Carla owes Bob 10 coins, Bob owes Alice 10 coins and Alice owes Carla 10 coins. This is documented by IOU assets on different shards S_{cb}, S_{ba} and S_{ac}. Each of the stakeholders only know their own balance. In particular, they are ignorant of the edge on their opposite side of the dependency triangle in Figure 5.8a. To explain how they will eventually create a transaction that does all the netting atomically we will first focus on Alice's point of view. She is willing to cancel the IOU that she has with Bob if the IOU Carla has with her is also canceled. So she collaborates with the other two individually to create to proposals for $\texttt{IOU}_{Bob \to Alice}$`.Forgive` and $\texttt{IOU}_{Alice \to Carla}$`.Forgive`. Alice would not mind if Carla simply forgave her debt. But, she needs to ensure that she does not forgive Bob's debt without Carla forgiving hers as well. This means from Alice's point of view a one-sided dependency exists between the two proposals. Because Alice's



(a) Illustration of business relations

Asset	Shard
$\text{IOU}_{Alice \to Carla}$	S_{ac}
$\text{IOU}_{Carla \to Bob}$	S_{cb}
$\text{IOU}_{Bob \to Alice}$	S_{ba}

(b) Location of the smart assets

Figure 5.8: Dependencies in a netting scenario

and Carla's IOU is stored on S_{ac}, she adds the dependency set $\{S_{ac}\}$ to the proposal for $\texttt{IOU}_{Bob \to Alice}\texttt{.Forgive}$ and leaves the dependency set for $\texttt{IOU}_{Alice \to Carla}\texttt{.Forgive}$ empty. However, Carla is in the same position, only she relies on Bob to forgive her debts before she can forgive Alice. So for her, $\texttt{IOU}_{Alice \to Carla}\texttt{.Forgive}$ must contain the dependency set $\{S_{cb}\}$. Equivalently, Bob adds the dependency set $\{S_{ba}\}$ to $\texttt{IOU}_{Carla \to Bob}\texttt{.Forgive}$. After all the negotiation is done, they end up with the following proposals:

$\texttt{IOU}_{Bob \to Alice}\texttt{.Forgive}$ – dependencies:$\{S_{ac}\}$

$\texttt{IOU}_{Alice \to Carla}\texttt{.Forgive}$ – dependencies:$\{S_{cb}\}$

$\texttt{IOU}_{Carla \to Bob}\texttt{.Forgive}$ – dependencies:$\{S_{ba}\}$

Note that when they combine the response hashes for these proposals into a transaction and sign it, then each one of them does not know one of the hashes. More importantly, they are unaware of the originating proposal. However, neither of the three needs to know about the agreement between the other two. As long as their own netting balances out they are satisfied. More importantly, they do not want any uninvolved party to know about their fiscal status.

During validation, the situation for the shards is similar. Each recognizes the proposal to forgive the debt tracked by their own internal state and they find they are dependent on the execution of one other shard. Yet, they never learn of the existence of the third part of the transaction. Still, all three parts will eventually commit atomically.

This example shows important features of TNG. It is possible for multiple stakeholders to collaborate on the creation of a transaction. Neither of these stakeholders needs to have complete knowledge of all the parts of a transaction. An additional proposal that is included into a transaction by another stakeholder cannot affect existing ones, because it operates on different smart assets. The addition of extra shards into a dependency set cannot break existing dependencies, only extend them. Therefore stakeholders can ignore any part of a transaction they do not care about and still be sure that their own parts will execute correctly and will not leak information.

We re-emphasize that TNG's transactions can be considered atomic, even though some parts of the transaction might be discarded by some shards, while others commit their parts. This happens *if and only if* these parts are not dependent on each other. In conventional database systems, atomic transactions guarantee that the system is never left in an inconsistent state. But in TNG, each part of a transaction manipulates a single smart asset and this state transition is already atomic. Therefore, each partial world state of a shard is always consistent, no matter what other shards do. The declaration of dependencies defines the *atomicity of intent*. TNG can only guarantee that cross-shard validation adheres to the dependencies that the stakeholders define. Because invalid hashes in a transaction are indistinguishable from hashes that belong to other shards, a shard cannot decide if dependencies are missing. To summarize, every part of a transaction on its own is atomic, all parts that are connected by dependencies are atomic, but in general it cannot be guaranteed that all parts of a transaction are connected by dependencies.

However, if it should be truly necessary for a TNG transaction to be committed or discarded as a whole, then we can weaken its privacy guarantees to achieve this: We can introduce the restriction that a transaction is only valid if the number of shards in the dependency set of a proposal is equal to the number of hashes minus one. This way, it is guaranteed that every hash belongs to a shard and each shard executes exactly one proposal. As a consequence, shards learn the full participation list and stakeholders cannot add bogus hashes to obfuscate the true number.

5.3.4 Summary of information flow

We now summarize the access to an asset's information to show that guarantee (IV), which states that unauthorized network participants can only learn private data if they are explicitly told by authorized participants, holds. As long as an asset is not interacted with, it remains in the partial world state of its custodian shard. It is not part of the partial world state of any other shard, so they cannot access it. Here, we emphasize the distinction between a node and a shard. Shards have completely disjoint views of the world state, they share no overlap. However, any node in the network can be part of multiple shards. As a consequence, a node can store assets spanning multiple shards in its local state database. This breaks no guarantees, because this node is explicitly trusted with all private data of all shards it belongs to. If a node that simultaneously belongs to shard A and shard B acts correctly, then there is no interaction in the transaction flow as described in Section 5.3.2 that would allow it to share private data belonging to shard A with any nodes that only belong to B and not A. Doing so breaks the assumption of our threat model, that is that nodes can be Byzantine but do not leak data.

In the context of transaction creation, only clients that are stakeholders of the asset can successfully get information from the shard that holds the asset. Furthermore, should a transaction include other clients that are not stakeholders of this asset, then they cannot learn anything, because the transaction itself contains only hashes and the private data remains on the clients and the shard. Once again the guarantee can only be broken by an authorized client disseminating data to unauthorized participants.

The validation protocol, which we will discuss in detail in Section 5.4, shares a transaction's hash and a single Boolean value that signifies the transaction's validity. This cannot be traced back to any of the shards' private assets. In particular, the validation would look identical if all parts of a transaction would be no-ops, i.e. a private state transition cannot be discriminated from doing nothing.

Importantly, this means that it is impossible for a malicious network participant to learn anything about an asset that it is not authorized to access. The only way the information can leak is the malicious dissemination by an authorized participant, which would break our threat model assumptions. Therefore, guarantee (IV) holds.

5.3.5 The necessity of global ordering

If two transactions are conflicting, they try to modify the same state. Therefore, we need a total order of *conflicting* transactions to create a deterministic ledger. Yet, to the

ordering service, transactions are indistinguishable, i.e. any transaction can potentially modify any local shard state and they do not themselves contain any information about the involvement of shards. Therefore, only the involved clients and shards know about potential conflicts. However, both clients and shards only have a partial view of the global ledger. So, even they cannot decide which transactions are conflict-free. Consequently, each transaction must be treated as conflicting with some other transaction and a total order of *all* transactions is needed. To totally order all transactions, the ordering service must have knowledge of all transactions, which by definition makes it a global service.

While it is theoretically possible to build a hierarchical global ordering service similar to the proposal by Amiri et al.[1], this would leak involvement information because specific ordering services would need to be assigned to specific shard combinations and we would need a power set of ordering services based on the number of all shards. Therefore, this is not feasible in practice.

5.3.6 Composability and business logic

From the clients' perspective the whole transaction flow is completely agnostic to the specific implementation of the underlying blockchain system. The shards' internal data structure is opaque to the clients. Furthermore, asset definitions are completely composable to form arbitrary business cases. There is no need to know what an asset will be used for in the future when it is initially designed. Only its own properties must be defined. This makes it possible for even a non-blockchain expert to build solutions on top of this protocol.

5.4 Privacy-preserving atomic commit protocol

During the validation step, each involved shard validates its own part of a transaction. Then the shards with dependent parts must come to an agreement about the validity of the transaction. Each shard relies on the validity reported by the other shards, because they have no access to their private data. Since the result for each shard is subjective due to a different view of the world state and a single discard overrules all other validity results, we can rely on a simpler atomic commit protocol instead of employing a full BFT consensus algorithm. In conventional database systems, the standard protocol is the two-phase atomic commit (2PC) protocol [11]. Here, all participants report their result to a coordinator node, which computes the final result based on all the inputs and reports back to the participants. Therefore, this protocol terminates after two rounds of communication.

However, our goal is to create a privacy-preserving atomic commit protocol `pCP` that only relies on minimal knowledge of the protocol participants.

To begin with, each shard has knowledge of the dependency set of its part of the transaction. Moreover, because it needs to contact the shards in the dependency set, those shards will also learn of its participation, if it is not already in their dependency set. In the following we will show how it is possible to create a protocol where this is the extent of participants that any given shard can learn and that is guaranteed to terminate and reach the correct conclusion.

To reason about the necessary properties of a privacy-preserving atomic protocol, we construct a dependency graph. In the dependency graph, each shard involved in the transaction becomes a vertex and dependencies between shards are given by directed edges as illustrated in Figure 5.10. For example, if shard s_1 depends on shard s_2 and that in turn depends on shard s_3, s_1 should not learn about the dependency on s_3. Still, all shards that are linked by dependencies must commit or discard a transaction atomically. To this end, we propose the protocol described by Algorithm 1 and Algorithm 2 and explain it in the following section.

Algorithm 1 `pCP`

function ValidateTx(*txHash, initialBelief, hashCount, deps*) ▷ *deps*: shard dependencies
  $b \leftarrow initialBelief$ ▷ *false = discard*, *true = commit*

  if Count(*deps*) $= 0 \ \vee \ b = false$ **then**
    StoreBelief(*txHash*, *b*, *isFinal*: true) ▷ save to internal storage beyond the sketch of this algorithm
    return *b*
  end if

  for *round* $\leftarrow 1,$ *hashCount* $-$ Count(*deps*) **do**
    StoreRound(*txHash*, *round*) ▷ save to internal storage
    StoreBelief(*txHash*, *b*, *isFinal*: false)
    $allFinal \leftarrow$ true

    for all *dep* **in** *deps* **do**
      ▷ remote call to other shard, caller made explicit for easier understanding
      b_{dep}, $isFinal \leftarrow dep$.PullBelief(*txHash*, thisShard)
      $allFinal \leftarrow allFinal \wedge isFinal$
      $b \leftarrow b \ \wedge \ b_{dep}$

      if not *b* **then**
        break all loops
      end if
    end for

    if *allFinal* **then**
      break loop
    end if
  end for

  StoreBelief(txHash, *b*, *isFinal*: true)
  return *b*
end function

Algorithm 2 `PullBelief` is called remotely by `pCP` on a different shard.

▷ function is called remotely from other shard, in practice $shard_{caller}$ would be obtained implicitly from caller address
function PULLBELIEF($txHash$, $shard_{caller}$)
    $i \leftarrow$ GETCALLCOUNT($txHash$, $shard_{caller}$) ▷ read from internal storage

    repeat
        $b, isFinal \leftarrow$ GETBELIEF($txHash$) ▷ read from internal storage
    until GETROUND($txHash$) $> i \vee isFinal$
        ▷ read from internal storage
    STORECALLCOUNT($txHash$, $shard_{caller}$, $i + 1$) ▷ save back to internal storage
    RESPONDTO($shard_{caller}$, b, $isFinal$) ▷ send back current belief
end function

5.4.1 Pseudocode description

Every shard acts both as a sever and client during the atomic commit protocol. For each transaction, the shard first validates it against its internal world state and labels it either valid or invalid. If the transaction stays valid throughout the protocol it will be committed, otherwise it will be discarded. Next, `ValidateTx` is called with several parameters.

- The transaction's hash $txHash$ is needed to identify the transaction with other shards.
- The belief to either *commit* or *discard* the transaction is based on the internal validation result and can change based on the feedback from other shards.
- The number of hashes that are part of the transaction are needed to be able to terminate the protocol in all edge cases. This will be further discussed in Section 5.4.3.
- The dependency set of this shard's part of the transaction is needed to contact the correct collaborating shards to achieve atomicity.

If the initial belief is to discard the transaction, then the shard simply stores this result in case other shards will ask for it and then ends its part of the protocol. To guarantee consitency, an invalid local part of a transaction can never be overruled by valid results from other shards.

Should the shard initially believe the transaction is valid, then it asks all its dependencies for their current belief of the validity by calling their `PullBelief` functions remotely. It then updates its own belief by ANDing all beliefs. This means that a single *discard* will overrule all *commit* messages. It repeats this process until either its belief switched to *discard* or it executed a number of these communication rounds equal to the difference between the number of hashes in the transaction and the number of shards this shard is dependent on. We show in Section 5.4.3 that this is an upper bound for guaranteed atomicity among all shards. Furthermore, as an optimization the shard can receive a flag together with the belief from a dependency that denotes that this dependency is sure it will not alter its belief anymore. If all dependencies send this *isFinal* signal, then the shard can stop the rounds of communication prematurely. A shard can be sure that its belief will not change anymore under three conditions:

1. Its internal belief is *discard*
2. It has no dependencies
3. All its dependencies sent the *isFinal* signal

Once its belief is finalized a shard can stop the protocol and either commit or discard the transaction.

Now we focus on the server side of the `PullBelief` function. Here, the called shard is asked for its belief for a specific transaction by other calling shards that depend on it. When a request arrives, as preparation for the next step, the called shard first checks how often the calling shard has already requested an update in its belief. It is an indicator which round the calling shard currently is in. Consider that the calling shard invokes `PullBelief` during each of its communication rounds exactly once for each dependency. Thus, with n previous calls the calling shard is currently in round $n + 1$.

Next, the called shard checks its own communication round. If the round number is currently lower than the number of calls and it has not finalized its own belief, it will wait in this step. It periodically check back to see if its own protocol advanced in communication rounds. Once its own round number is larger than the number of previous calls by the calling shard, it advances and checks its current internal belief. At this point it is guaranteed that it is at least in the same round as the caller if not ahead, or its own internal belief is finalized. This acts as a weak synchronisation of communication rounds: The calling shard cannot proceed until it has received a response from the called shard and the called shard can catch up with its own communication rounds. If the called shard responded immediately, the caller would receive the same response as after a previous

request. What is worse, the calling shard could not differentiate between a stale response and an updated belief that happened to have the same value as before. However, as we show in the following section, counting the steps of information propagation is crucial to enable shards to reason about the state of the system. This asynchronous callback model creates a link between the round number and the information propagation an therefore makes belief finalization possible for every dependency configuration, as we prove in Theorem 2 in Section 5.4.3. Therefore, shards can rely on the round number as a signal for protocol termination. By responding with a same value without update based on new input, this assumption would be violated. The calling shard would progress through its protocol without learning new information and terminate prematurely.

After it ensured that the calling shard is not ahead in the protocol, the called shard increments its request counter and responds to the calling shard. Note that its own `ValidateTx` function is executed concurrently, so its own round counter can rise while it stalls the response to the other shard.

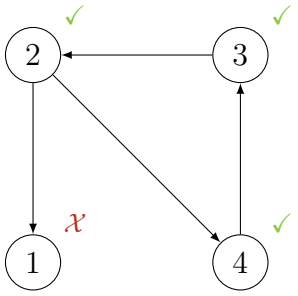
We illustrate this algorithm with an example in Figure 5.9. Initially, only shard 1 believes the transaction to be invalid. Each round, every shard communicates with its dependencies, i.e. it calls the shard the arrow in the dependency graph points to. This way shard 1's belief is propagated to the next shard. After four rounds, all shards have updated their belief and discard the transaction. Note how the belief travels in the reverse direction of shard dependencies. Even though shard 4 is a neighbour of shard 2, it only receives the information after four rounds, because it only ever queries shard 3. Trivially, the number of steps for a state to propagate across the network is upper bounded by the network diameter, which in the worst case is n, for a linear network.

5.4.2 Preserving privacy

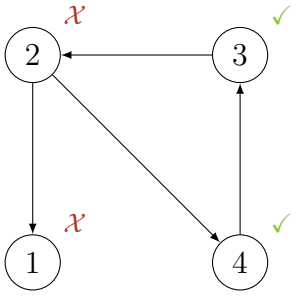
Our proposed protocol is a generalization of classic atomic commit protocols with all-to-all communication replaced by a (sparse) communication overlay in the form of the dependency graph of the transaction. Examples of such dependency graphs are illustrated in Figure 5.10.

Involved shards start the protocol only knowing their own dependencies on other shards. Theoretically, they have three options for how to communicate their local validation result: Broadcast it to all nodes, send it to a single coordinator or contact the known shards in the dependency set.

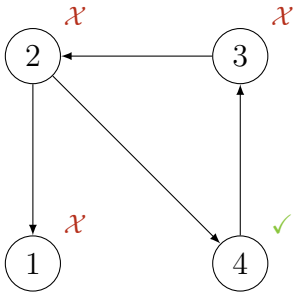
In the first case, the whole network would learn about the involvement of specific shards, contradicting the goal of achieving privacy. In the second case, the coordinator would learn



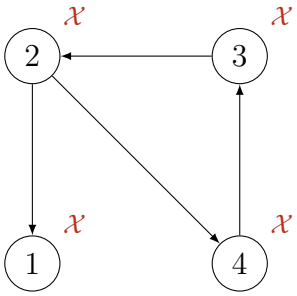
Round 0: Initial validation results



Round 1: Shard 2 changes its belief. Shard 3 and shard 4 continue the protocol.

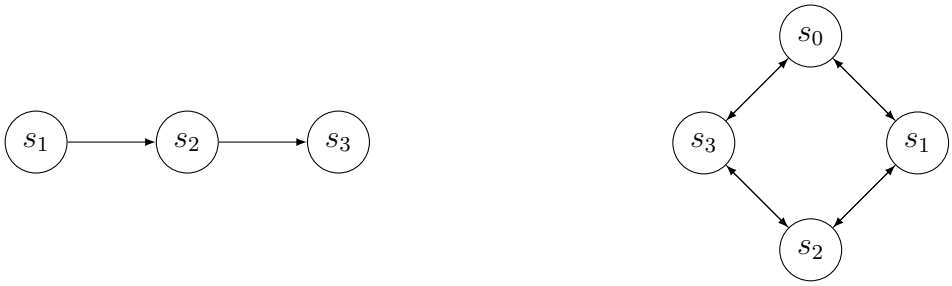


Round 2: Shard 3 changes its belief. Shard 4 continues the protocol.



Round 3: Shard 4 changes its belief. All shards have terminated.

Figure 5.9: Example of an atomic commit with `pCP`. Each shard is only aware of its direct neighbours.



(a) Transitively directed dependency: s_1 does not learn about s_3 and vice versa; s_3 can decide commitment without consulting the other shards.

(b) Mutual dependency: Every shard is dependent on all its neighbours and cannot commit without communication; no shard learns about the shard on the opposing corner of the square.

Figure 5.10: Examples of dependency graphs

the identity of all involved shards and make it a central point of attack. This makes the third option the preferred one from a privacy standpoint, because each shard only learns of its direct neighbours in the dependency graph and no shard has complete knowledge of the communication overlay (unless it is fully connected).

Using the dependency graph as the communication overlay still guarantees that every shard learns all necessary data because transitive dependencies form paths in the graph. Therefore, a shard is connected to another shard if and only if it is (transitively) dependent on that shard.

Furthermore, by absorbing learned results from other shards into their own internal validation beliefs, shards can make the dependency paths opaque to anyone but their direct neighbours.

However, it is insufficient to simply ask the neighbours in the dependency graph for their belief once and then commit based on the result. Imagine a scenario with four shards s_0, s_1, s_2, s_3. Let each shard s_i depend on the result from two shards $s_{i-1 \mod 4}$ and $s_{i+1 \mod 4}$ as shown in Figure 5.10b. If s_1 is valid and the responses from s_0 and s_2 are valid as well, it would go ahead and commit. But it could be the case that s_1 asked before either of the other shards had a chance to get a response from s_3. If s_3 is invalid, both s_0 and s_2 would change their result to invalid as well. Therefore, s_1 should not have committed its part of the transaction. This means multiple rounds of communication are necessary. In the following, we will prove bounds on the number of necessary rounds and show that termination is guaranteed.

5.4.3 Bounds on communication rounds

For the purposes of this discussion, we define a round of communication as all shards finishing one iteration of the outer for loop in Algorithm 1. This presents a logical batching rather than a temporal one, since shards are only loosely synchronized by the `WaitForBeliefUpdate` command. Yet, under the assumption of a partially synchronous system there is a bounded delay between two shards executing the same round of communication. In the following, we examine how many rounds of communication are necessary for shards to atomically commit a given transaction.

Lemma 1. *Each round of communication can only relay information along the edges of the dependency graph. Any information can only travel the length of one edge per round.*

Proof. Assume that a given shard s_i can communicate with a shard s_j, where s_i and s_j are not directly connected by the dependency graph. By necessity, s_i would need to learn about s_j, even though s_j is not in s_i's dependency set and vice versa. This is explicitly forbidden. □

Lemma 2. *For any two given shards s_i and s_j there exists a finite or countably infinite set of paths between them.*

Proof. The number of edges of the dependency graph is finite, otherwise the underlying transaction would be infinitely large. Therefore we can enumerate them all. For the purposes of this proof we choose a counting system with a base at least as large as the number of edges. This way, each edge is labeled by a unique symbol.

Starting from s_i we create all possible paths by traversing all connecting edges and writing down the symbols of the edges. Then, for each vertex we landed on, we repeat this process, appending the new edge number to the previous sequence. The process for a given path ends when s_j is reached. Since the dependency graph can have cycles, sequences can be arbitrarily long. But, all paths from s_i to s_j must be finite by definition. Therefore, we can sort the sequences belonging to those paths in ascending order and count them. □

Lemma 3. *For any two given connected shards s_i and s_j it takes exactly l rounds of communication to relay information from one to the other, where l is the length of the shortest path between the shards. If there is no path, then $l = \infty$.*

Proof. As by Lemma 2, the paths between two shards are countable, so we can order all paths between s_i and s_j by their length. Now, we let all shards relay all their information to

all neighbours in every round of communication. According to Lemma 1, the information can make progress along each path one edge at a time. We can track this by subtracting 1 from the length of each path. After l rounds the shortest path will have 0 length, indicating that s_j received the information.

There cannot be a shorter path that is not on the list, because by definition we listed all possible paths between s_i and s_j.

If there is no path in the ordered list, then this search will never stop, i.e. $l = \infty$. □

Theorem 1. *Let* `pCP` *be an algorithm that ensures atomic commitment of a transaction tx while keeping the global set of participants secret. Then,* `pCP` *must go through at least l^* rounds of communication, where l^* is the length of the longest* shortest path *between any two connected shards of the dependency graph for tx.*

Proof. For a transaction to commit correctly, every shard must commit or discard the transaction atomically together with all other dependent shards. By necessity, a shard must learn all validation results of the shards it is transitively dependent on because a single `discard` signal would change its own belief. According to Lemma 3, for every shard s_i, every shortest path to another connected shard s_j must be traversed. If two shards are not connected, they do not require to learn each other's belief, so for all relevant paths is $l < \infty$. The transaction is not finalized before the last one of them reaches its destination. Therefore, the longest length of all shortest paths forms the lower bound on finalization. □

Note that this is only a lower bound on global transaction finalization. Single shards are able to finalize their beliefs in the following four cases:

- It discarded the transaction in its own validation.
- It is not dependent on any other shard.
- It received a `discard` signal from one of its collaborating shards.
- All of its collaborating shards signal that they finalized their belief.

A finalized shard must broadcast its belief to its neighbours before dropping out of the commit protocol.

If shards do not encounter any of these cases, they might not stop after l^* rounds, because they are not able to construct the full dependency graph. Therefore, l^* is not known to the shards.

Lemma 4. *For a given shard s_i, it's longest* shortest path *to another connected shard s_j has at most length $n - \delta^+(s_i)$, where n is the number of hashes in the transaction and $\delta^+(s_i)$ is the out-degree of vertex s_i.*

Proof. For s_i, $\delta^+(s_i)$ shards are connected by a shortest path of length 1. This leaves at most $n - \delta^+(s_i) - 1$ unknown shards[4]. In the worst case they are positioned as a chain with $n - \delta^+(s_i) - 2$ edges connecting them. This chain of shards must be connected to s_i via one of its direct neighbours. Let s_j be the last vertex in the chain. Then, the shortest path from s_i to s_j is

$$\begin{aligned} l^* &= \underbrace{1}_{\text{neighbour}} + \underbrace{1}_{\text{connect chain}} + (n - \delta^+(s_i) - 2) \\ &= n - \delta^+(s_i). \end{aligned}$$

□

Theorem 2. *All shards finalize their belief after at most $n - \min_i \delta^+(s_i)$ rounds of communication.*

Proof. Directly follows from Lemma 4. □

5.5 Experiments

We now evaluate a proof-of-concept implementation of the simplified privacy protocol based on Hyperledger Fabric v1.4. In this context, each peer of the network represents a distinct shard. The theoretical extension to a fully decentralized setting is described in Section 5.6. We opted to make our modifications as unobtrusive as possible. In this vein, our modified version TNG-Fabric retains all of Fabric's features, with the added capability of processing private transactions created by our protocol. To this end, custom code that is completely independent of Fabric handles the creation of private transaction proposals described in Section 5.3.2. Then, it calls a wrapper around Fabric's endorsers that unpacks the message and translates it into a Fabric transaction proposal, which in turn is forwarded to regular Fabric endorsers. The response is directed back through the wrapper, where it is stored together with the proposal. In particular, the installed chaincodes in this example translate smart assets into key-value pairs. While this does not adhere to the design principles we

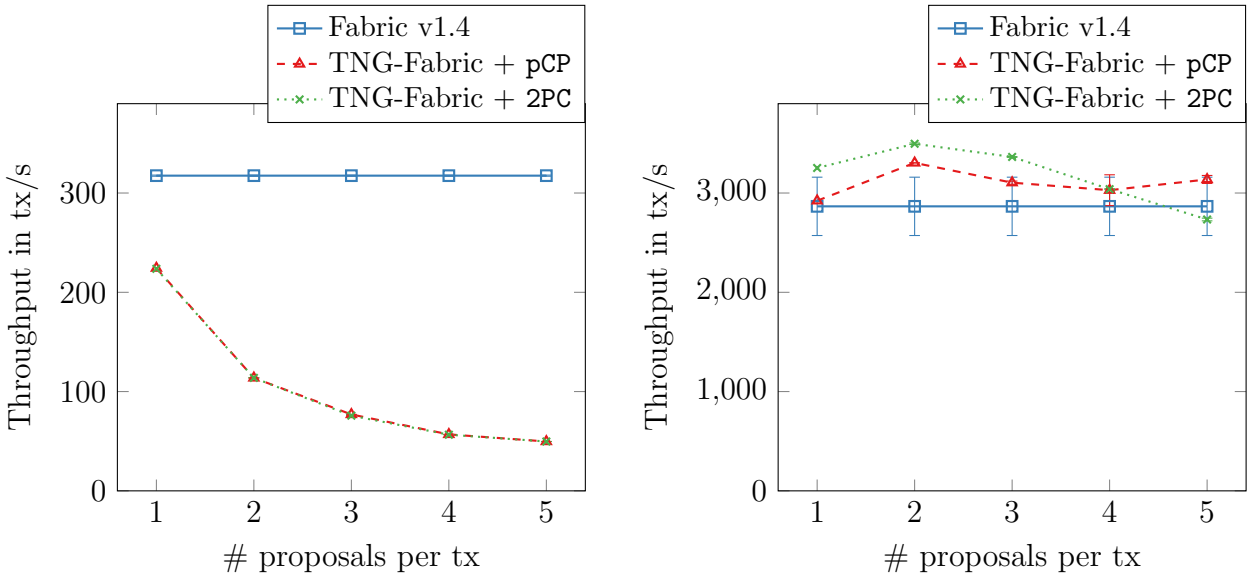
[4]Some of the hashes could be associated with the same shard or be invalid.

outlined in Section 5.2.3, it makes it easier to directly compare the world state changes between Fabric and its TNG extension. Only a hash is returned to the client according to Section 5.3.2. The ordering service of Fabric stays unchanged as it does not unpack the contents of a transaction. Lastly, we added a new flag `PRIVATE_TRANSACTION` to the grpc messages Fabric uses for communication. This way, a peer can correctly deserialize a private transaction during validation, look up the stored response and engage in the `pCP` from Section 5.4.

5.5.1 Experimental setup

For our performance analysis we implemented a similar chaincode to the described toy example in section 5.3.2: Each shard manages a number of accounts with their own balances and each transaction proposal asks to move some amount of coins from one account to another. A transaction then consists of hashes that act as anchors to multiple such proposals. Each proposal reaches a different shard and we vary the dependencies between them for each experiment as we will describe in the following sections. Regular Fabric chaincode transferring coins between two accounts reads the current values of two accounts and then writes back the updated values. We align one-proposal TNG transactions with this case, so we can estimate the impact of the privacy protocol more easily, instead of obfuscating the results by comparing different data management workloads. This means, each TNG proposal also reads from and writes to two keys in Fabric's state database. We stress that this tightly couples the smart asset definition to Fabric's data store and only serves to facilitate the evaluation of our experimental results. To externally verify correct execution after an experimental run, the modified account balances on each shard change by a different amount. This way, we can easily check the final balance on the peers to ensure that each peer executed the correct proposals.

For the experiments we use nine local servers: one client, three orderers forming a Raft cluster and five endorsing peers. Each of these processes is spawned in its own Docker container and runs on a server equipped with two Intel® Xeon® CPU E5-2620 v2 processors at 2.10 GHz, for a total of 24 hardware threads and 64 GB of RAM. We record the overall transaction throughput of TNG-Fabric and the latency of the `pCP` protocol. We compare the performance of TNG-Fabric with regular Fabric v1.4 and a version of TNG-Fabric in which we substituted `pCP` with a two-phase commit protocol (`2PC`). Furthermore, we ran all experiments with just-in-time endorsements and pre-endorsed transaction, where all transactions are created before the test run and then immediately submitted to the ordering service. This allows us to gauge the performance impact of the protocol on



(a) just-in-time endorsed transactions

(b) pre-endorsed transactions

Figure 5.11: Impact of private transaction creation and pCP on Fabric's transaction throughput

the endorsement and validation step independently. Each experimental run sends 10,000 transactions to the network.

5.5.2 Throughput

First, we evaluate how much overhead our privacy modifications impose on the transaction throughput. In particular, we are interested in measuring the impact of adding more proposals to a transaction on the end-to-end throughput of the system, i.e. including all steps from endorsement to commitment, as well as just on the validation and commitment throughput. The results of these experiments are presented in Figure 5.11.

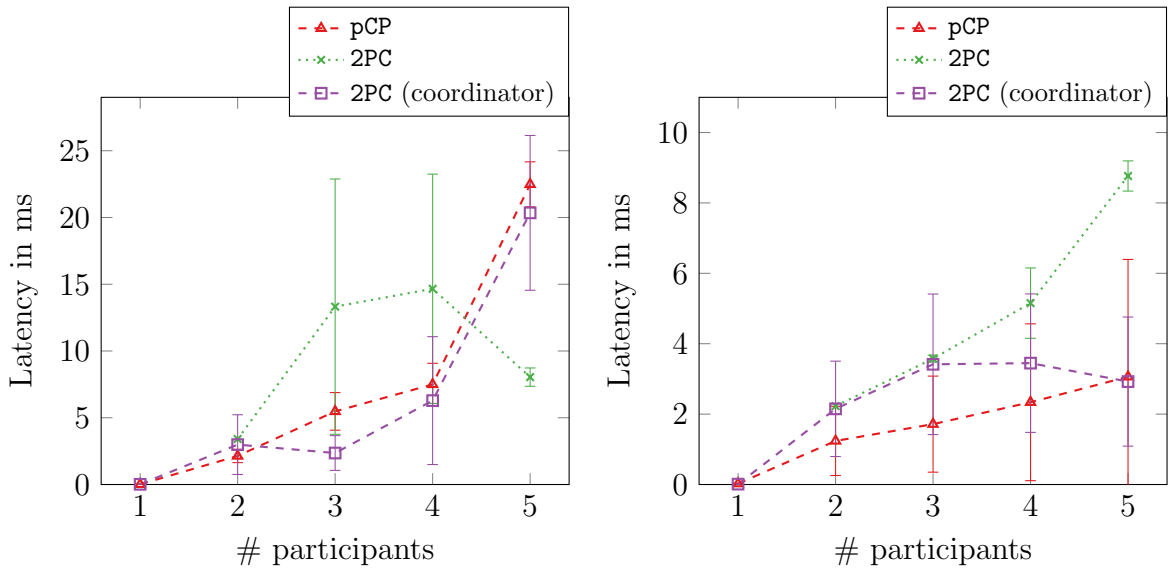
Each transaction involves one to five proposals, each targeting a different shard. Those proposals are linked sequentially into a dependency chain like in Figure 5.10a. We start with the investigation of the impact on the end-to-end throughput shown in Figure 5.11a. For a single proposal, which produces the same RW set as a regular Fabric transaction, we see a performance hit of about 30%. We find the throughput decreases roughly proportional

to the number of proposals, i.e. for n proposals the total throughput sinks to $\frac{1}{n}$ of the original value. This is not surprising, since our proof-of-concept client interacts with the endorsers sequentially. Effectively, a TNG-Fabric transaction with n proposals translates to n regular Fabric transactions. This means our experiments show that the number of affected world state keys across all shards stays close to 230 per second. Furthermore, we see nearly identical results regardless of the used commit protocol during validation. The negligible impact of the commit protocol on performance is due to the fact that we use XOX's dependency analyzer as described in Section 4.3 to identify conflict-free transactions. Then, we start the commit protocol concurrently for all transactions that don't have any state conflicts with each other. Therefore, the commit protocol is not a blocking operation and throughput is unimpacted.

This finding is corroborated by the experiments with pre-endorsed transactions which focus on the validation and commitment throughput of shards (Figure 5.11b). Once the calls to endorse proposals are excluded from the measurements, all tested systems show very similar performance. Regular Fabric seems to even be outperformed by TNG-Fabric. However, this is due to TNG-Fabric skipping the validation of the endorsement policy that regular transactions go through. In this case the endorsement policy for regular Fabric was set to accept any endorser, therefore TNG skips a single signature check per transaction. Because TNG transactions have a different structure than regular Fabric's transactions, it would have meant a complete reimplementation of the endorsement validation. This was beyond the scope of this proof of concept. With chaincode execution not being the dominant factor anymore, the graph also shows that there is a slight overhead to pay for the added privacy of `pCP` over regular two-phase commit in the shard communication. We conclude that endorsement of a private transaction proposal imposes a constant overhead of about 30% on the transaction throughput, but with the help of a dependency analyzer the impact of privacy-preserving validation is negligible.

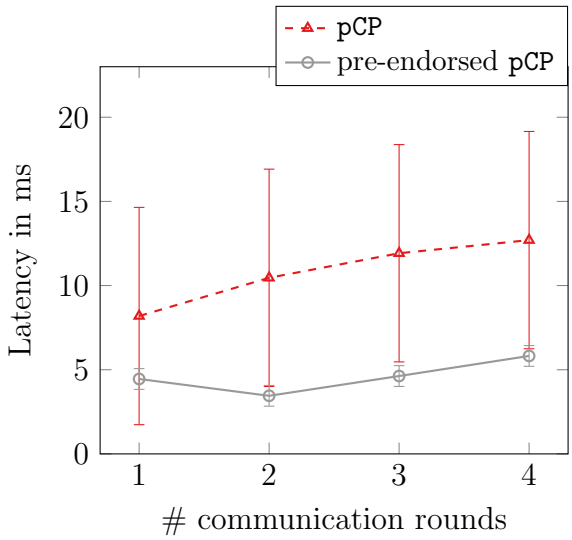
5.5.3 Latency

Next, we investigate the performance difference between our proposed privacy-preserving atomic commit protocol (`pCP`) and a conventional two-phase commit (`2PC`). To this end, we measured the latency of the inter-shard validation step, which lets us compare the time spent in each protocol. Because `pCP` adopts a less efficient communication overlay in favor of preserving participant anonymity, our initial assumption is that it is outperformed by (`2PC`). In Figures 5.12a and 5.12b we show the latency depending on the number of participants in the commit protocol for a given transaction. In the case of `2PC`, all peers with a proposal belonging to the respective transaction contact a single coordinator, who



(a) just-in-time endorsed transactions

(b) pre-endorsed transactions



(c) transactions with cyclic dependencies

Figure 5.12: Latency comparison of our privacy-preserving commit protocol (pCP) and the two-phased commit protocol (2PC)

decides the outcome based on the input of the other peers and then disseminates the result back to them. For pCP, we measure the latency of a transaction with a chain dependency between proposals. This means that peer_0 will be able to decide immediately, because it has no dependencies, peer_1 is dependent on the outcome of peer_0, peer_2 is dependent on peer_1 and therefore transitively dependent on peer_0 and so on. Therefore, peer_n will go through n rounds of pCP.

Surprisingly, we find almost identical behaviour of a peer with n dependencies in pCP and the coordinator communicating with n participants in 2PC. Additionally, we encounter vastly diverging latencies from the non-coordinator participant in 2PC. Especially for the case of 3 or 4 protocol participants, we found that some nodes took systematically more time to terminate than others. With pre-endorsed transactions, the pCP even outperforms 2PC and both cut their latencies in half, even though endorsement should have no impact on this measurement. We attribute these unexpected results to the block dissemination mechanism of Fabric. The time to completion of one peer is dependent on the timings when its dependent peers have received the same block and start the commit protocol. Here, 2PC is more strongly affected, because peers can only make progress when all other peers have communicated with the coordinator. Most likely, some peers received blocks consistently earlier than others and then had to wait for those to catch up. In contrast, a peer using pCP is unaffected by the status of peers further up in the dependency chain. In fact, even in the reverse case, where a peer is dependent on many other peers, it might be able to take advantage of its dependent peers having already finalized their conclusions, so it can stop the protocol early. This means, even though 2PC theoretically commits in fewer rounds than pCP in cases with long chain dependencies, in practice this does not hurt the performance of TNG-Fabric.

To measure the communication overhead more rigorously, we must ensure that peers cannot terminate the protocol early. To this end, in our final experiment we investigate the latency of pCP with cyclic dependencies as illustrated in Figure 5.10b. In this case, the peers cannot cut the rounds short because they have no knowledge of the size of the dependency cycle, apart from the upper bound we proved in Section 5.4.3. As Figure 5.12c shows, the latency for added communication rounds is below 2ms. Even when we enforce four rounds of communication, the protocol completes in well under 20ms on our localized cluster.

5.5.4 Implications

Our experiments show that the transaction creation, including proposal simulation on the endorsers, has the biggest impact on performance by a large margin. However, we

can easily improve this by parallelizing this step. While it cannot be as fast as Fabric's equivalent because it takes more rounds of communication, this would eliminate the $\frac{1}{n}$ trend in the throughput for n proposals per transaction and instead only have a constant factor overhead. Next, we demonstrated that `2PC` is not significantly more efficient than `pCP`. Our protocol benefits from the fact that it requires a weaker temporal synchronisation between protocol participants than `2PC`. In the case of two protocol participants, which we believe is the most common one, `pCP` and `2PC` show almost the same performance. Furthermore, it is evident that, at least in a setting with localized shards nodes, the cross-shard validation does not impose a significant overhead to Fabric's throughput. We conclude that a careful implementation of the TNG protocol could come close to the performance of a fully public Fabric blockchain. This is almost five times faster than the execution of private contracts on JP Morgan Chase's Quorum blockchain [6], which has a less capable privacy model than TNG (Section 5.7).

5.6 Generalizing TNG

Up to this point we have described TNG as an extension to Hyperledger Fabric in a simplified black-box environment. In this section, we make two generalizations. First, we show that a large range of blockchain systems could implement TNG. Second, we describe the necessary change to run TNG in a fully decentralized setting

5.6.1 Arbitrary permissioned blockchains

Because shards only respond with hashes to proposals and state transitions are only fully realised after they are committed, the execution engine and data management of the underlying blockchain is completely decoupled from the transaction flow. Shards could be configured to either do pre-order (XO) or post-order execution (OX) or some mix of the two (XOX). Our description of the transaction flow based on Fabric already covered any blockchain that follows an XO model. It can be easily extended to work with XOX by introducing a secondary execution step after the internal transaction validation analogously to Chapter 4.

Moreover, TNG also allows post-order execution with some minor tweaks to the node. When a client submits a proposal to a shard it produces a hash based on the proposal and its simulation result. However, in the OX model, there is no chaincode simulation. If the shard would simply sign the proposal and send it back, then this would open a

vector of attack to learn the private key of a shard by repeatedly feeding the shard with specially prepared proposal messages. Therefore, we require the shard to produce a nonce and use it as a substitute for the simulation result. This way, it is still guaranteed that the hash is generated by the shard. Then, when a shard recognizes a familiar hash in a transaction, it loads the proposal from its transient store and executes it. In contrast to regular OX blockchains, it cannot immediately commit the result. First it needs to engage in the cross-shard validation using `pCP`. After that step it can finalize the state transition and continue with the next transaction.

5.6.2 Full decentralization

After describing the protocol in the simplified case with the ordering service and the privacy shards being replaced by black boxes, we now explain the necessary extensions to make it work in a fully decentralized scenario.

The ordering service is a completely isolated component of the protocol, so we can replace the black box with any consensus protocol that meets the assumed threat model. Because we need a global order of transactions, we rely on a high throughput consensus algorithm. Giving the additional benefit of being byzantine fault tolerant, we favor Mir-BFT [110] for this task.

For the generalization of shards we look at its interfaces. A shard interacts with clients before the transaction is submitted just like a Fabric peer would, so we can employ endorsement policies to combat faulty behaviour. Next, it receives new blocks from the ordering service. Assuming f nodes in a shard can be faulty, the ordering service needs to send new blocks to at least $f + 1$ nodes so that they can broadcast the blocks to the rest of the shard. Lastly, we have the interface to other shards during cross-shard validation. The easiest solution for this is to run a shard internal consensus algorithm which collects $f + 1$ signatures for each validation result and attaches this to the validation response messages. We can define f implicitly by the number of peers per shard $n = 3f + 1$ [69]. This way, f does not need to become a configured parameter of the network.

Putting a consensus protocol in the critical path of the validation can have a large impact in performance. To mitigate this, care must be taken to localize nodes of the same shard so that network communication between nodes is fast and use a dependency analyzer to parallelize transaction validation in order to amortize the extra latency introduced by the consensus.

5.7 Related Work

Fabric's channels act as completely isolated blockchain instances, even when using the same hardware nodes in the network. This makes them a potential tool to preserve privacy. However, there is no way of interaction between channels. Androulaki et al. [4] explored possibilities of cross-channel transactions to exchange assets. Among their proposed solution they also discuss a privacy-preserving cross-channel transaction which makes use of tear-off Merkle trees to exchange necessary proof for validation. However, this protocol requires participants to go through three steps: locking the assets that should get exchanged on their local private channels, then exchanging the cross-channel transaction to proof the exchange and then finally unlocking the asset on the new channel. In this work, we propose a protocol that allows cross-shard transactions that resolve in a single step.

CAPER [1] models the blockchain as a directed acyclic graph (DAG). This enables sharding of transactions, which is used both to increase data privacy for the ledger as well as scaling performance. Each shard maintains an internal private chain of transaction that is intertwined with a global public cross-shard chain. Both internal chains and the global chain are totally ordered. This approach works well for scenarios with tightly siloed data pockets that can easily be sharded and where cross-shard transactions are rare. In this case, internal transactions of different shards can be executed in parallel. However, if the workload does not have clear boundaries to separate the shards, then most transactions will use the global chain, negating the benefit of CAPER.

While there are increasing efforts such as by Kosba et al. [62] and Bünz et al. [16], it is still not possible to create a framework based on zero-knowledge proofs that can deal with arbitrary applications and generating proofs takes minutes even on high performance machines. Quorum, an enterprise blockchain based on Ethereum, substitutes transaction payload with hashes and only grants authorized parties access to the payload in a similar fashion as TNG [94]. However, it has several shortcomings. It is not possible to create transactions that span multiple privacy regions like TNG's cross shard transactions. Malicious nodes can stall the system or create inconsistent states. They can submit a transaction hash to the ledger and then time the dissemination of the payload, so that some nodes receive it and others do not. Lastly, it is almost five times slower than TNG [6].

Brandenburger et al. [14, 13] propose to move smart contract execution to an *enclave* inside a Trusted Execution Environments (TEEs) such as Intel's SGX platform. While this makes the chaincode execution tamper-proof, they face the challenge that a TEE can be fed with any arbitrary input by a malicious peer that operates the TEE. In particular,

the peer can control the world state that the chaincode operates on. Even if the world state is encrypted, the peer can learn information by repeatedly triggering smart contract execution while slightly altering the input. Because TEE's do not store an internal state, they are susceptible to such replay attacks. This can be solved naively by moving the whole peer instance into the trusted environment, but this contradicts Intel's developer guidelines and contradicts the best practice of minimizing the trusted computing base (TCB), which in turn minimizes the attack surface of a system. Their solution introduces a second enclave which runs a verifier for the most recent world state. This way, the chaincode can call this verifier to ensure that the world state it is being fed has not been reset to a previous time and in fact represents the correct and most recent state. Trusted execution is orthogonal to our work presented in this chapter. In particular, it can significantly increase the security of our privacy-preserving atomic commit protocol described in Section 5.4 in a fully Byzantine environment.

Starting with the two-phase atomic commit protocol (2PC), the first and simplest of the atomic commit protocol (ACP) family, these algorithms have been studied for decades. They ensure that a distributed system either collectively commits or aborts an operation. Chrysanthis et al. [23] give an extensive overview over existing algorithms. Generally, they either improve on 2PC by making the happy path more efficient or by making the algorithm more resilient against failures. In particular, the variant of *non-blocking* ACPs allows correct nodes to make progress without waiting for any failed nodes. To the best of our knowledge we are the first to investigate the angle of preserving privacy in the context of atomic commit protocols.

Practically all current blockchain systems work the same way, in that they rely either on built-in functionality like Bitcoin's coin UTXO model or require predefined smart contracts to be uploaded to the ledger. The idea of splitting assets from their interaction has recently been explored by NEM [82] and Prasaga [92]. However, they still treat assets as descriptive pieces of information. Instead, we enrich assets with the ability to govern their own state, effectively introducing a kind of object-oriented programming to the blockchain world. Ciatto et al. [24] propose the use of a logic language such as Prolog instead of Ethereum's Virtual Machine (EVM) to define smart contracts. Each smart contract gets associated with a *goal*, the description of the desired effect in case of a valid invocation. Additionally, they attach a *static* knowledge base, a set of immutable rules and terms, as well as a dynamic knowledge base with mutable structures to smart contracts. These rules govern the validity of a transaction. Hereby, the dynamic knowledge base can be used to modify the validation rules over time. This strict separation of validity rules and computation effect is designed to help developers reason about smart contract execution and make smart contracts more amenable to change over time. However, it is not explained how

exactly these new data structures would be handled by the blockchain's data management system. In addition to the introduction of a logic-based programming paradigm, this work proposes inter-smart contract interactions, i.e. composability of smart contracts, by creating a similar model to Fabric's current chaincode-to-chaincode links. We discussed the shortcomings of this approach in Section 5.1.2.

5.8 Limitations and Future Work

In this chapter, we presented a novel privacy protocol for permissioned blockchain systems. It enables collaboration between companies without the need to make their sensitive private data public. We use the insight that sharding of the blockchain's world state is not only useful for enhanced performance but also to allow to control data access. Our protocol allows atomic cross-shard transactions without leaking any information of the content. We built this on top of a flexible programming model which replaces smart contracts with smart assets. Software developers can easily compose them to create complex business cases and need no knowledge of the underlying blockchain system. In this section we present the limitations of this protocol and directions for future work.

5.8.1 Verifiably correct cross-shard transactions

Cross shard transactions are composed by stakeholders. While the system guarantees that they can never create a transaction that transitions a shard into an inconsistent state, the protocol itself does not prevent nonsensical compositions. For example, it is possible to create a transaction that withdraws 10 coins from an account on one shard and deposits 20 coins to another account on a different shard. To mitigate this, a trusted third party could be used as an additional stakeholder in each part of the transaction to act as an auditor. However, this belies the point of implementing a blockchain system in the first place. In the future, we could explore the possibility of shards returning a certified simulation result which can be used as input for other parts of the transaction. This would mean that a money deposit would require a certificate of a withdrawal of an equal amount from some other account. This somewhat weakens the privacy guarantees of the TNG protocol, so special care must be taken to minimize the amount of revealed information.

5.8.2 Public data

TNG assumes all data is private and stored by shards. It can simulate a public blockchain by adding every single node to an all-encompassing shard, but this would incur an overhead compared to a simple public ledger because nodes would need to disseminate the transaction content separately from the ledger *before* the transaction is submitted to guarantee all nodes recognize the hashes on the ledger. To address this, we could reintroduce the concept of public transactions. Thus, the content of these special transactions would not be replaced by hashes so that they could get disseminated together with the blocks. In fact, our proof of concept already implicitly allows this. To take advantage of existing implementation as much as possible, we simply added a new transaction type and modified the validation and commitment pipeline when necessary. This means that our TNG-Fabric implementation still understands regular public transactions. As another extension, a future version of the protocol could allow public and private parts in the same transaction.

5.8.3 Decentralized shards

While we described the general idea how TNG can be decentralized from the simplified black-box approach, the actual implementation is non-trivial. For example, it must be guaranteed that all nodes in a shard see the pre-image of a proposal, before the corresponding transaction is submitted to the ordering service. This would introduce the need for a reliable atomic broadcast protocol inside the shard for data dissemination. Even though it is entirely possible to generalize TNG to a decentralized setting, due to the drastically higher complexity of the system all edge cases must be investigated thoroughly to guarantee data consistency and prevention of information leakage.

5.8.4 Development framework

TNG introduces a flexible programming model that makes data storage and retrieval entirely the blockchain system's responsibility and allows stakeholders to compose transactions through collaborative negotiation. To effectively take advantage of this flexibility, software developers need tools to easily create interactive workflows for transaction creation. Furthermore, smart assets are now distributed among many separate shards. To create a thriving ecosystem, developers need a way to discover installed asset definitions.

5.8.5 Trusted Execution Environments

In its current form, TNG requires stakeholders to trust shards with their private data. TNG cannot prevent malicious nodes from disseminating sensitive information if they are authorized to store it. With the inclusion of Trusted Execution Environments (TEE) we could eliminate this problem. In such a system, private data would be stored completely encrypted so that malicious nodes could only spread ciphertext. Then, when a proposal needs to be executed, this is done in a TEE. The TEE loads the encrypted data, decrypts it and manipulates the smart asset according to the proposal, then stores the result back into the transient store after encrypting it. At no time would a node have access to the private data's cleartext.

Chapter 6

Conclusion

Blockchain systems uniquely enable collaborative applications through their decentralized and trustless nature. While permissionless systems suffer from weak performance and lack of governance, permissioned blockchains promise to bridge the gap for enterprises to move away from conventional solutions like distributed databases towards a cooperative future. This thesis focuses on eliminating current pain points with permissioned blockchains, using Hyperledger Fabric as a prototyping platform.

6.1 Contributions

Our contributions span from performance optimization in the general case as well as workload dependent to building a framework for permissioned blockchains which allows for fine-grained control of privacy and an additional layer of abstraction for potential blockchain developers.

We re-engineered Hyperledger Fabric to support nearly 20,000 transactions per second, a factor of almost 7 better than prior work. We accomplished this goal by implementing a series of independent optimizations focusing on I/O, caching, parallelism and efficient data access. In our design, orderers only receive transaction IDs instead of full transactions, and validation on peers is heavily parallelized. We also use aggressive caching and we leverage light-weight data structures for fast data access on the critical path.

We proposed a novel hybrid execution model for Hyperledger Fabric consisting of a pre-order and a post-order execution step. This allows a trade-off between parallel transaction execution and minimal invalidation due to conflicting results. In particular, our solution

can deal with highly skewed workloads where most transactions use only a small set of hot keys. Contrary to other post-order execution models, we support the use of external oracles in our secondary execution step. We show that the throughput of our implementation scales comparably to Fabric and FastFabric for low contention workloads, and surpasses them when transaction conflicts increase in frequency.

Lastly, we introduce a privacy-preserving blockchain framework without complex cryptographic algorithms by relying on sharding and targeted dissemination of information. Shards form units of shared views of a partial world state. Therefore, shards can be used to model information flow in collaborative business cases. Our protocol allows a shared global ledger without leaking information to unauthorized participants. Our main insight is that the knowledge about the nature of a collaboration lies with the clients of the ledger. Therefore, transaction validation can be split into two parts: clients agree on the content and craft transactions with the necessary dependency information while the shards only need to check that their world state stays consistent. This is supported by a novel atomic commit protocol that can rely on incomplete knowledge of the participants and provably arrives at a deterministic conclusion in a finite amount of time. The way we compartmentalize transactions for privacy also greatly increases composability of smart contracts as a side effect. Instead of representing all the business logic, smart contracts in our protocol only describe the properties of the assets they create. These assets present public APIs which software developers can use to interact with the blockchain without the need for expert blockchain knowledge. We showed with a proof of concept that our privacy protocol has performance comparable to a version of Hyperledger Fabric that incorporates no privacy mechanisms.

6.2 Future Work

We see multiple directions in which the performance of Hyperledger Fabric can be further improved. Now that we showed that the peer throughput is able to catch up and even overtake the ordering service, it would be interesting to incorporate an efficient BFT consensus algorithm such as RCanopus [60] or Mir-BFT [110]. Our profiling results lead us to believe that the peer performance can be even further improved by focusing on the cryptographic validation of transactions. We see two interesting ways forward. First, the existing cryptographic computation library could be replaced with a more efficient one. Second, we could evaluate scaling the cryptographic validation of transactions across multiple servers. By intelligently sharding independent world state namespaces, we could make use of Fabric's channels and scale the network horizontally by effectively parallelizing multiple identical

blockchains. Additionally, we left the exploration of data analytics on top of the storage peer for future work. Here, an efficient data analytics layer using a distributed framework such as Apache Spark could be implemented.

For higher throughput under contentious workloads we believe the largest gain will be in the creation of a high-performance secondary execution step, since this showed the largest overhead in our experiments. Furthermore, it would be useful to explore ways in which the smart contract code for the first and second execution steps could be unified to facilitate development and reduce the risk for software bugs.

For our privacy protocol, future efforts should be directed into speeding up the private transaction creation and providing easy-to-implement interfaces for users of arbitrary blockchain applications. Also, instead of creating wrappers around Fabric to simulate the protocol, the functionality should be directly built in to reduce computational overload. This requires the replacement of Fabric's chaincode execution engine with a new execution environment specifically tailored for smart assets. Lastly, a BFT network could be tested, where shards are formed of groups of peers which execute localized BFT consensus when communicating with other shards.

6.3 Closing Thoughts

Today's tech sector is dominated by single companies controlling access to virtual ecosystems. With the blockchain technology, it can become possible to usher in a paradigm shift away from a dependence on these companies towards a more collaborative environment. To make this a feasible reality, blockchains must be able to scale in a similar matter as centrally controlled distributed systems currently do. On top of that, they need to offer real advantages over the status quo.

In this dissertation, we addressed the scalability issues of the popular permissioned blockchain system Hyperledger Fabric and created a framework that allows both fine-grained access control to private data and unrestricted interaction between collaborators on the platform. Lastly, the introduction of a programming model that is decoupled from the blockchain's data management allows software developers to move to this new paradigm more easily. We hope that the improvements described in this thesis can herald the next generation of permissioned blockchain systems which will spawn novel collaborative applications.

References

[1] Mohammad Javad Amiri, Divyakant Agrawal, and Amr El Abbadi. CAPER: a cross-application permissioned blockchain. *Proceedings of the VLDB Endowment*, 12(11):1385–1398, 2019.

[2] Mohammad Javad Amiri, Divyakant Agrawal, and Amr El Abbadi. ParBlockchain: Leveraging Transaction Parallelism in Permissioned Blockchain Systems. *Proceedings - International Conference on Distributed Computing Systems*, 2019-July:1337–1347, July 2019.

[3] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolić, Sharon Weed Cocco, and Jason Yellick. Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains. *Proceedings of the Thirteenth EuroSys Conference on - EuroSys '18*:1–15, 2018.

[4] Elli Androulaki, Christian Cachin, Angelo De Caro, and Eleftherios Kokoris-Kogias. Channels: horizontal scaling and confidentiality on permissioned blockchains. In *European Symposium on Research in Computer Security*, pages 111–131. Springer, 2018.

[5] Arati Baliga, Nitesh Solanki, Shubham Verekar, Amol Pednekar, Pandurang Kamat, and Siddhartha Chatterjee. Performance Characterization of Hyperledger Fabric. In *Crypto Valley Conference on Blockchain Technology, CVCBT 2018*, 2018.

[6] Arati Baliga, I Subhod, Pandurang Kamat, and Siddhartha Chatterjee. Performance Evaluation of the Quorum Blockchain Platform. *arXiv preprint arXiv:1809.03421*, July 2018.

[7] Shehar Bano, Alberto Sonnino, Mustafa Al-Bassam, Sarah Azouvi, Patrick McCorry, Sarah Meiklejohn, and George Danezis. Consensus in the age of blockchains. *arXiv preprint arXiv:1711.03936*, 2017.

[8] Christian Kobhio Bassek, Samuel Pierre, and Alejandro Quintero. Redundancy Schemes for High Availability Computer Clusters. *Journal of Computer Science*, 2(1):33–47, January 2006.

[9] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: decentralized anonymous payments from bitcoin. In *IEEE Symposium on Security & Privacy*, pages 459–474, 2014.

[10] Sara Bergman, Mikael Asplund, and Simin Nadjm-Tehrani. Permissioned blockchains and distributed databases: A performance study. *Concurrency and Computation: Practice and Experience*:e5227, 2018.

[11] Philip A Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*, volume 370. Addison-Wesley Longman Publishing Co., Inc., 1987.

[12] Alysson Bessani, João Sousa, and Eduardo E.P. Alchieri. State machine replication for the masses with BFT-SMaRt. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 355–362, 2014.

[13] Marcus Brandenburger and Christian Cachin. Challenges for combining smart contracts with trusted computing. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 20–21, New York, New York, USA. Association for Computing Machinery, October 2018.

[14] Marcus Brandenburger, Christian Cachin, Rüdiger Kapitza, and Alessandro Sorniotti. Blockchain and Trusted Computing: Problems, Pitfalls, and a Solution for Hyperledger Fabric. *arXiv preprint arXiv:1805.08541*, May 2018.

[15] Ethan Buchman. *Tendermint: Byzantine Fault Tolerance in Age of Blockchain*. Master's thesis, University of Guelph, 2016.

[16] Benedikt Bünz, Shashank Agrawal, Mahdi Zamani, and Dan Boneh. Zether: Towards Privacy in a Smart Contract World. *IACR Cryptology ePrint Archive*, Report 201:191, 2019.

[17] Vitalik Buterin. What Proof of Stake Is And Why It Matters, 2013. URL: https://bitcoinmagazine.com/articles/what-proof-of-stake-is-and-why-it-matters-1377531463/.

[18] Jan Camenisch, Manu Drijvers, and Anja Lehmann. Anonymous attestation using the strong diffie hellman assumption revisited. In *Trust and Trustworthy Computing*, pages 1–20. Springer Verlag, 2016.

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

[20] Centers for Disease Control and Prevention. Outbreak of E. coli Infections Linked to Romaine Lettuce, 2019. URL: https://www.cdc.gov/ecoli/2019/o157h7-11-19.

[21] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *7th Symposium on Operating Systems Design and Implementation (OSDI '06), November 6-8, Seattle, WA, USA*:205–218, 2006.

[22] Si Chen, Jinyu Zhang, Rui Shi, Jiaqi Yan, and Qing Ke. A comparative testing on performance of blockchain and relational database: Foundation for applying smart technology into current business systems. In *International Conference on Distributed, Ambient, and Pervasive Interactions*, pages 21–34. Springer Verlag, 2018.

[23] Panos Chrysanthis, George Samaras, and Yousef Al-Houmaily. Recovery and Performance of Atomic Commit Processing in Distributed Database Systems. In *Recovery Mechanisms in Database Systems*, pages 370–416. Prentice-Hall, Inc., 1998.

[24] Giovanni Ciatto, Roberta Calegari, Stefano Mariani, Enrico Denti, and Andrea Omicini. From the Blockchain to Logic Programming and Back: Research Perspectives. In *CEUR Workshop "From Objects to Agents" Proceedings*, pages 69–74, 2018.

[25] Cloud Native Computing Foundation. gRPC: A high performance, open-source universal RPC framework, 2017. URL: https://grpc.io/.

[26] Susan B. Davidson, Hector Garcia-Molina, and Dale Skeen. Consistency in Partitioned Network. *ACM Computing Surveys (CSUR)*, 17(3):341–370, September 1985.

[27] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. Monkey : Optimal Navigable Key-Value Store. *Proceedings of the 2017 international conference on Management of Data - SIGMOD '17*:79–94, 2017.

[28] Primavera De Filippi and Samer Hassan. Blockchain technology as a regulatory technology: From code is law to law is code. *First Monday, special issue on 'Reclaiming the Internet with distributed architectures'*, 21(12), December 2016.

[29] Jeffrey Dean and Sanjay Ghemawat. LevelDB: A Fast Persistent Key-Value Store, 2011. URL: https://opensource.googleblog.com/2011/07/leveldb-fast-persistent-key-value-store.html.

[30] Edsger W. Dijkstra. On the Role of Scientific Thought. In *Selected Writings on Computing: A personal Perspective*, pages 60–66. Springer New York, 1982.

[31] Tien Tuan Anh Dinh, Ji Wang, Gang Chen, Rui Liu, Beng Chin Ooi, and Kian-Lee Tan. BLOCKBENCH: A Framework for Analyzing Private Blockchains. *Proceedings of the 2017 ACM International Conference on Management of Data - SIGMOD '17*:1085–1100, 2017.

[32] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)*, 35(2):288–323, April 1988.

[33] Ian Aragon Escobar, Eduardo E.P. Alchieri, Fernando Luís Dotti, and Fernando Pedone. Boosting concurrency in Parallel State Machine Replication. In *Proceedings of the 20th International Middleware Conference*, pages 228–240. Association for Computing Machinery (ACM), 2019.

[34] Victoria Espinel, Derek O'Halloran, Erik Brynjolfsson, and Domhnall O'Sullivan. Deep shift: Technology tipping points and societal impact. In *New York: World Economic Forum–Global Agenda Council on the Future of Software & Society (REF 310815)*, 2015.

[35] Ethereum Community. EthHub Documentation – Proof of Stake, 2019. URL: https://docs.ethhub.io/ethereum-roadmap/ethereum-2.0/proof-of-stake/.

[36] Ethereum Community. EWASM, 2018. URL: https://github.com/ewasm.

[37] Ethereum Community. Go Ethereum, 2013. URL: https://github.com/ethereum/go-ethereum.

[38] Martin Fowler. Event Sourcing, 2005. URL: https://martinfowler.com/eaaDev/EventSourcing.html.

[39] Patrick D. Gallagher, Cameron F. Kerry, and Charles Romine. *FIPS PUB 186-4: Digital Signature Standard (DSS)*. NIST, 2013.

[40] Google Developers. Protocol Buffers, 2008. URL: https://developers.google.com/protocol-buffers.

[41] Christian Gorenflo. Personal discussion with the BigChain development Team, 2018.

[42] Christian Gorenflo, Stephen Lee, Lukasz Golab, and Srinivasan Keshav. FastFabric: Scaling Hyperledger Fabric to 20,000 Transactions per Second. *IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*:455–463, May 2019.

[43] Christian Gorenflo, Stephen Lee, Lukasz Golab, and Srinivasan Keshav. FastFabric: Scaling Hyperledger Fabric to 20,000 Transactions per Second. In *International Journal of Network Management*. John Wiley and Sons Ltd, February 2020.

[44] Ye Guo and Chen Liang. Blockchain application and outlook in the banking industry. *Financial Innovation*, 2(1):24, December 2016.

[45] Suyash Gupta, Sajjad Rahnama, Jelle Hellings, and Mohammad Sadoghi. ResilientDB: Global Scale Resilient Blockchain Fabric. *arXiv preprint arXiv:2002.00160*, February 2020.

[46] Justus Haucap and Ulrich Heimeshoff. Google, Facebook, Amazon, eBay: Is the Internet driving competition or market monopolization? *International Economics and Economic Policy*, 11(1-2):49–61, February 2014.

[47] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, January 1990.

[48] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. ZooKeeper: Wait-free Coordination for Internet-scale Systems. *USENIX annual technical conference*, 8(9), 2010.

[49] Hyperledger Fabric. Documentation. URL: https://hyperledger-fabric.readthedocs.io.

[50] Hyperledger Fabric. Documentation – Private data, 2019. URL: https://hyperledger-fabric.readthedocs.io/en/release-1.4/private-data/private-data.html.

[51] Hyperledger Fabric. Hyperledger Fabric: github repository, 2019. URL: https://github.com/hyperledger/fabric.

[52] Hyperledger Fabric. Hyperledger JIRA – [FAB-12221] Validator/Committer refactor. URL: https://jira.hyperledger.org/browse/FAB-12221.

[53] Hyperledger Fabric. Source Code. URL: https://github.com/hyperledger/fabric.

[54] IBM. IBM Blockchain – Food Trust, 2018. URL: https://www.ibm.com/blockchain/solutions/food-trust.

[55] IBM. TradeLens, 2018. URL: https://www.tradelens.com/.

[56] Azadeh Jahanbanifar, Ferhat Khendek, and Maria Toeroe. Providing hardware redundancy for highly available services in virtualized environments. In *Proceedings - 8th International Conference on Software Security and Reliability, SERE 2014*, pages 40–47. Institute of Electrical and Electronics Engineers Inc., 2014.

[57] Haris Javaid, Chengchen Hu, and Gordon Brebner. Optimizing validation phase of hyperledger fabric. In *Proceedings - IEEE Computer Society's Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems, MASCOTS*, volume 2019-Octob, pages 269–275, July 2019.

[58] Elena Karafiloski and Anastas Mishev. Blockchain solutions for big data challenges: A literature review. In *17th IEEE International Conference on Smart Technologies, EUROCON 2017 - Conference Proceedings*, pages 763–768. Institute of Electrical and Electronics Engineers Inc., August 2017.

[59] Ghassan O. Karame, Elli Androulaki, and Srdjan Čapkun. Double-spending fast payments in Bitcoin. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 906–917, New York, New York, USA. ACM Press, 2012.

[60] Srinivasan Keshav, Wojciech Golab, Bernard Wong, Sajjad Rizvi, and Sergey Gorbunov. RCanopus: Making Canopus Resilient to Failures and Byzantine Faults. *arXiv preprint arXiv:1810.09300*, October 2018.

[61] Sunny King and Scott Nadal. PPCoin: Peer-to-Peer Crypto-Currency with Proof-of-Stake, 2012. URL: https://www.peercoin.net/whitepapers/peercoin-paper.pdf.

[62] Ahmed Kosba, Andrew Miller, Elaine Shi, Zikai Wen, and Charalampos Papamanthou. Hawk: The Blockchain Model of Cryptography and Privacy-Preserving Smart Contracts. In *Proceedings - 2016 IEEE Symposium on Security and Privacy, SP 2016*, pages 839–858. Institute of Electrical and Electronics Engineers Inc., August 2016.

[63] Jay Kreps, Neha Narkhede, and Jun Rao. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*, volume 11, pages 1–7, 2011.

[64] Avinash Lakshman and Prashant Malik. Cassandra - A decentralized structured storage system. In *Operating Systems Review (ACM)*, volume 44 of number 2, pages 35–40, April 2010.

[65] Leslie Lamport. The implementation of reliable distributed multiprocess systems. *Computer Networks (1976)*, 2(2):95–114, May 1978.

[66] Leslie Lamport. The Part-Time Parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.

[67] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.

[68] Leslie Lamport. Using Time Instead of Timeout for Fault-Tolerant Distributed Systems. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 6(2):254–280, 1984.

[69] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, July 1982.

[70] Yoad Lewenberg, Yoram Bachrach, Yonatan Sompolinsky, Aviv Zohar, and Jeffrey S. Rosenschein. Bitcoin mining pools: A cooperative game theoretic analysis. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems, AAMAS*, 2015.

[71] Loren Muchnick. Can I run more than 1024 containers on a single Docker engine?, 2017. URL: https://success.docker.com/article/maximum-containers-per-engine.

[72] Fumio Machida, Masahiro Kawato, and Yoshiharu Maeno. Redundant virtual machine placement for fault-tolerant consolidated server clusters. In *Proceedings of the 2010 IEEE/IFIP Network Operations and Management Symposium, NOMS 2010*, pages 32–39. IEEE Computer Society, 2010.

[73] David Malone and Karl .J. O'Dwyer. Bitcoin Mining and its Energy Footprint. In *25th IET Irish Signals & Systems Conference 2014 and 2014 China-Ireland International Conference on Information and Communities Technologies (ISSC 2014/CIICT 2014)*, pages 280–285. Institution of Engineering and Technology, 2014.

[74] MemcacheDB. MemcacheDB: A distributed key-value storage system designed for persistent, 2008. URL: https://web.archive.org/web/20180809162514/http://memcachedb.org/.

[75] Francisco Memoria. The Flippening: Is Ethereum Set to Become the #1 Cryptocurrency?, 2017. URL: https://www.ccn.com/flippening-ethereum-set-become-1-cryptocurrency/.

[76] Ralph Charles Merkle. A Digital Signature Based on a Conventional Encryption Function. In *Advances in Cryptology — CRYPTO '87*, pages 369–378. Springer, 1988.

[77] Ralph Charles Merkle. *Secrecy, authentication, and public key systems.* PhD thesis, Stanford University, 1979.

[78] Martin Moore and Damian Tambini. *Digital Dominance: the Power of Google, Amazon, Facebook, and Apple*. Oxford University Press, 2018, page 423.

[79] Satoshi Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System, 2008. URL: https://bitcoin.org/bitcoin.pdf.

[80] Qassim Nasir, Ilham A Qasse, Manar Abu Talib, and Ali Bou Nassif. Performance analysis of hyperledger fabric platforms. *Security and Communication Networks*, 2018, 2018.

[81] Pezhman Nasirifard, Ruben Mayer, and Hans-Arno Jacobsen. FabricCRDT: A Conflict-Free Replicated Datatypes Approach to Permissioned Blockchains. In *Proceedings of the 20th International Middleware Conference*, pages 110–122. Association for Computing Machinery (ACM), 2019.

[82] NEM. Documentation, 2019. URL: https://docs.nem.io/en.

[83] Katherine Nield and Ricardo Pereira. Fraud on the European Union emissions trading scheme: Effects, vulnerabilities and regulatory reform. *European Energy and Environmental Law Review*, 20(6):255–289, 2011.

[84] Nxt Wiki. White Paper, 2016. URL: https://nxtwiki.org/wiki/Whitepaper:Nxt.

[85] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (LSM-tree). *Acta Informatica*, 33(4):351–385, June 1996.

[86] Michael a Olson, Keith Bostic, and Margo Seltzer. Berkeley DB. *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, 1999.

[87] Diego Ongaro and John Ousterhout. In Search of an Understandable Consensus Algorithm. *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, 22(2):305–320, 2014.

[88] Marshall Pease, Robert Shostak, and Leslie Lamport. Reaching Agreement in the Presence of Faults. *Journal of the ACM (JACM)*, 27(2):228–234, April 1980.

[89] Suporn Pongnumkul, Chaiyaphum Siripanpornchana, and Suttipong Thajchayapong. Performance analysis of private blockchain platforms in varying workloads. In *2017 26th International Conference on Computer Communications and Networks, ICCCN 2017*, pages 1–6. IEEE, July 2017.

[90] Joseph Poon and Vitalik Buterin. Plasma : Scalable Autonomous Smart Contracts Scalable Multi-Party Computation, 2017. URL: https://plasma.io/plasma.pdf.

[91] Joseph Poon and Thaddeus Dryja. The Bitcoin Lightning Network: Scalable Off-Chain Instant Payments, 2016. URL: https://www.bitcoinlightning.com/bitcoin-lightning-network-whitepaper/.

[92] Prasaga. XSOA Whitepaper, 2019. URL: https://www.prasaga.com/xsoa-whitepaper/.

[93] go-python. Python 3.4 interpreter implementation for Golang. URL: https://github.com/go-python/gpython.

[94] Quorum. Documentation – Private State, 2019. URL: http://docs.goquorum.com/en/latest/#publicprivate-state.

[95] Pandian Raju, Rohan Kadekodi, and Ittai Abraham. PebblesDB : Building Key-Value Stores using Fragmented Log-Structured Merge Trees. *Proceedings of the 26th Symposium on Operating Systems Principles*:497–514, 2017.

[96] Ravi Kiran Raman, Roman Vaculin, Michael Hind, Sekou L Remy, Eleftheria K Pissadaki, Nelson Kibichii Bore, Roozbeh Daneshvar, Biplav Srivastava, and Kush R Varshney. Trusted multi-party computation and verifiable simulations: a scalable blockchain approach. *arXiv preprint arXiv:1809.08438*, 2018.

[97] Kai Ren, Qing Zheng, Joy Arulraj, and Garth Gibson. SlimDB : A Space-Efficient Key-Value Storage Engine For Semi-Sorted Data. *Proceedings of the VLDB Endowment*, 10(13):2037–2048, 2017.

[98] Eric Rescorla and Tim Dierks. The transport layer security (TLS) protocol version 1.3. *RFC 8446*, 2018.

[99] Reuters. Amazon under EU antitrust fire over use of merchant data - Reuters, 2019. URL: https://www.reuters.com/article/us-eu-amazon-com-antitrust-idUSKCN1UC0W9.

[100] Ronald L. Rivest, Adi Shamir, and Leonard Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978.

[101] Sajjad Rizvi, Bernard Wong, and Srinivasan Keshav. Canopus: A Scalable and Massively Parallel Consensus Protocol. In *Proceedings of the 13th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '17, pages 426–438, New York, NY, USA. ACM, 2017.

[102] Pingcheng Ruan, Gang Chen, Tien Tuan Anh Dinh, Qian Lin, Dumitrel Loghin, Beng Chin Ooi, and Meihui Zhang. Blockchains and Distributed Databases: a Twin Study. *arXiv preprint arXiv:1910.01310*, October 2019.

[103] Francisco Santos and Vasileios Kostakis. *The DAO: a million dollar lesson in blockchain governance*. Master's thesis, Tallin University of Technology, 2018.

[104] Fred B. Schneider. Byzantine generals in action: Implementing fail-stop processors. *ACM Transactions on Computer Systems (TOCS)*, 2(2):145–154, May 1984.

[105] Fred B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319, 1990.

[106] Ankur Sharma, Felix Martin Schuhknecht, Divyakant Agrawal, and Jens Dittrich. Blurring the Lines between Blockchains and Database Systems. In *Proceedings of the 2019 International Conference on Management of Data*, pages 105–122, 2019.

[107] Alessandro Sorniotti, Angelo De Caro, Baohua Yang, Binh Nguyen, Manish Sethi, Vukolic Marko, Sheehan Anderson, Srinivasan Muralidharan, and Parth Thakkar. Fabric Proposal: Enhanced Concurrency Control, 2017. URL: https://docs.google.com/document/d/1Z37O9nbpqBmukZQ88r2MmCr_DzTez--mCV5m3awhP-U.

[108] João Sousa, Alysson Bessani, and Marko Vukolić. A byzantine Fault-Tolerant ordering service for the hyperledger fabric blockchain platform. In *Proceedings - 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2018*, pages 51–58, 2018.

[109] V. Srinivasan, Brian Bulkowski, and Rajkumar Iyer. Aerospike : Architecture of a Real-Time Operational DBMS. *Proceedings of the VLDB Endowment*, 9(13):1389–1400, 2016.

[110] Chrysoula Stathakopoulou, Tudor David, and Marko Vukolić. Mir-BFT: High-Throughput BFT for Blockchains. *arXiv preprint arXiv:1906.05552*, June 2019.

[111] Douglas R. Stinson. Some observations on the theory of cryptographic hash functions. *Designs, Codes, and Cryptography*, 38(2):259–277, February 2006.

[112] Tendermint Foundation. Tendermint Core, 2017. URL: https://tendermint.com/core/.

[113] Parth Thakkar, Senthil Nathan, and Balaji Vishwanathan. 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, 2018.

[114] University of Cambridge. Cambridge Bitcoin Electricity Consumption Index (CBECI). URL: https://www.cbeci.org/.

[115] Hoang Tam Vo, Sheng Wang, Divyakant Agrawal, Gang Chen, and Beng Chin Ooi. LogBase: A Scalable Log-structured Database System in the Cloud. *Proceedings of the VLDB Endowment*, 5(10):1004–1015, June 2012.

[116] Marko Vukolić. The quest for scalable blockchain fabric: Proof-of-work vs. BFT replication. In *International Workshop on Open Problems in Network Security*, volume 9591, pages 112–125, 2016.

[117] Gavin Wood. Ethereum: a Secure Decentralised Generalised Transaction Ledger, 2014. URL: https://ethereum.github.io/yellowpaper/paper.pdf.

[118] Gavin Wood. Polkadot: Vision for a Heterogeneous Multi-Chain Framework, 2017. URL: https://polkadot.network/PolkaDotPaper.pdf.

[119] Congcong Ye, Guoqiang Li, Hongming Cai, Yonggen Gu, and Akira Fukuda. Analysis of security in blockchain: Case study in 51%-attack detecting. In *Proceedings - 2018 5th International Conference on Dependable Systems and Their Applications, DSA 2018*, pages 15–24. Institute of Electrical and Electronics Engineers Inc., December 2018.

[120] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. HotStuff: BFT Consensus with linearity and responsiveness. In *Proceedings of the Annual ACM Symposium on Principles of Distributed Computing*, pages 347–356, New York, New York, USA. Association for Computing Machinery, July 2019.

[121] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark : Cluster Computing with Working Sets. *HotCloud'10 Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, 10(10-10):10, 2010.

[122] Shenbin Zhang, Ence Zhou, Bingfeng Pi, Jun Sun, Kazuhiro Yamashita, and Yoshihide Nomura. A Solution for the Risk of Non-deterministic Transactions in Hyperledger Fabric. *IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*:253–261, 2019.