UNIVERSIDAD POLITÉCNICA
DE MADRID

ESCUELA TÉCNICA SUPERIOR DE INGENIEROS INFORMÁTICOS

MÁSTER UNIVERSITARIO EN INGENIERÍA DEL SOFTWARE –
EUROPEAN MASTER IN SOFTWARE ENGINEERING

Implementation of a Low-Latency Distributed Ledger

Master Thesis

Lucas Kuhring

Madrid, July 2019

This thesis is submitted to the ETSI Informáticos at Universidad Politécnica de Madrid in partial fulfillment of the requirements for the degree of Master of Science in Software Engineering.

Master Thesis
Master Universitario en Ingeniería del Software – European Master in Software Engineering
Thesis Title: Implementation of a Low-Latency Distributed Ledger
Thesis no: EMSE-2019-3
July 2019

Author: Lucas Kuhring
Master of Science
Universidad Politécnica de Madrid

Supervisor:

Manuel Carro Liñares
Associate Professor
Universidad Politécnica de Madrid

Co-supervisor:

Zsolt István
Assistant Research Professor
IMDEA Software Institute

Lenguajes y Sistemas Informáticos e
Ingeniería de Software
E.T.S. de Ingenieros Informáticos
Universidad Politécnica de Madrid

ETSI Informáticos
Universidad Politécnica de Madrid
Campus de Montegancedo, s/n
28660 Boadilla del Monte (Madrid)
Spain

Abstract

Thanks to public cryptocurrencies, blockchain systems have gained increasing popularity in the recent years. They rely on the processing of transactions in blocks, a way of amortizing large communication delays and costly consensus in the form of proof-of-work. Permissioned blockchain systems are usually neither geo-distributed, nor require proof-of-work, and yet, adopt the processing at block granularity, which makes them suffer from high latencies.

On top of Hyperledger Fabric, we address this problem by treating transactions as a stream instead of blocks. At the same time we implement batching mechanisms under the hood to amortize the costs of disk writes, and introduce further improvements to maintain high throughput. Concretely, our implementation manages to decrease the end-to-end latency of Fabric to the order of 1 ms, while keeping the throughput at a high rate of more than 3000 transactions/s.

Resumen

Gracias a las criptomonedas públicas, los sistemas de blockchain han ganado una gran popularidad durante los últimos años. Estos sistemas realizan el procesamiento de transacciones en bloques como una manera de amortizar altos delays en la comunicación y el coste del consenso en forma de proof-of-work. Los sistemas de blockchain autorizados normalmente no están geográficamente distribuidos ni requieren proof-of-work y, a pesar de ello, adoptan el procesamiento de transacciones en bloques, lo que les hace tener latencias altas.

Basándonos en Hyperledger Fabric afrontamos este problema tratando las transacciones como un stream en lugar de como bloques. Al mismo tiempo implementamos mecanismos de batching internos para amortizar el coste de las escrituras a disco, e introducimos mejoras adicionales para mantener un alto throughput. Concretamente, nuestra implementación logra disminuir la latencia end-to-end de Fabric en el orden de 1 ms, mientras mantiene el throughput a una alta velocidad mayor que 3000 transacciones/s.

INDEX

List of Figures

Chapter 1

Introduction

The notion of blockchain systems, also called distributed ledgers, is often associated with public cryptocurrencies like Bitcoin, which implement a very specific data model for the exchange of digital funds. Systems like Ethereum [1] or NEO [2], often referred to as the next generation of blockchains, enable more freedom and let their users create own data models and programs, so called smart contracts, to be executed upon the ledger. In its core they usually still rely on digital currencies, like Ether in the case of Ethereum, or NEO tokens in the same-named system. The latency of blockchain systems, being the time from when a transaction is issued, until it is part of the common ledger, is usually very high, talking about several minutes in the case of Bitcoin. The reason for that is the usage of proof-of-work as a consensus mechanism, which needs the participants to solve a complex cryptographic puzzle before they can publish a batch of transactions (block) to the network.

Next to the public blockchains, there exist private (permissioned) ones like Hyperledger Fabric [3] or Quorum [4], consisting only of a defined list of nodes and a trusted third party determines the permissions of the participating nodes, allowing consensus mechanisms that are more efficient and well-known from, e.g., distributed databases. Although there is no artificial latency added by proof-of-work and byzantine-fault-tolerant (BFT) algorithms can be performed in the order of milliseconds [5], permissioned ledgers are still slow, restraining their widespread adoption in practice.

In this work, we look at the case of Hyperledger Fabric (or short "Fabric"), where we can observe latencies of several 100 milliseconds. The reason is that an ordering service groups transactions into blocks, which are validated at the side of each peer. In public blockchains, blocks are used in order to reduce the costs of proof-of-work and make the communication over a geo-distributed more efficient. Although these assumptions generally don't apply for permissioned ledgers, they inherit this design from their public counterparts.

Fabric lets us configure the number of transactions inside a block, so that the waiting time of transactions inside the ordering service can be avoided by defining single-transactions blocks. However, as seen in Figure 1.1, small blocks come with the draw-

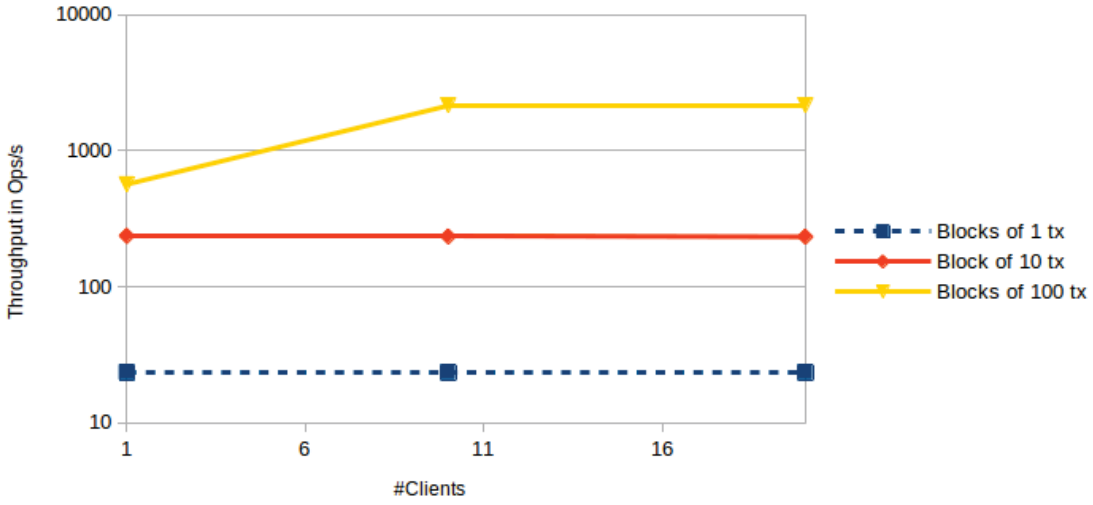

Figure 1.1: Throughput drops in case of single-transaction blocks

back of a low throughput. The graph shows on a logarithmic scale how the latter evolves with an increasing number of clients issuing transactions and therefore increasing the load on the system. Next to falling throughput, latencies for 1-transaction-blocks are still in the order of 100 milliseconds.

We propose a new design for blockchains and with StreamChain, implement it on top of Fabric, introducing the processing of transactions in a stream instead of blocks. We integrate practices from stream processing and overlap communication and computation, as well as introduce parallelism at multiple levels of the transaction processing inside a peer. We further implement the selective batching of transactions when writing them to the persistent ledger, amortizing expensive disk accesses. Additionally, we introduce further improvements under the hood, like caching block contents that were unmarshalled after transferring them on the network.

Thanks to our modifications, we achieve significantly decreased end-to-end latencies in the order of a millisecond and maintain high throughput of >3000 Ops/s. As a side effect, our solution turns out to decrease the percentage of failing transactions at the validation stage and therefore effectively increases the "goodput" of the system.

The first part of this work gives a more detailed introduction into the context of blockchains and explain the components and mechanics of Hyperledger Fabric. In the next chapter, we analyze and discuss Fabrics performance characteristics. The design and proof-of-concept implementation of StreamChain, together with our additional improvements are addressed in Chapter 5, followed by a performance evaluation of our implementation in the following chapter. We conclude and give ideas about future work in Chapter 6.

Chapter 2

Background

This chapter gives an overview of the concepts discussed in this work. We explain the term of blockchain systems and line out the characteristics of permissioned blockchain systems among that domain. In the next part, we address the related concept of smart contracts. Defining the application of the system. they are essential components inside of blockchain systems. As a particular instance of permissioned ledgers, we look at Hyperledger Fabric, addressing its architecture and working. Finally, we give a brief overview of related work on the benchmarking and improvement of performance of the Fabric platform.

2.1 Blockchain Systems

Blockchains, or also referred to as distributed ledgers, are append-only data structures maintained within a network of potentially untrusting peers. They consists of a log of transactions grouped together into blocks, which are cryptographically linked, forming a hash chain. Every block saves the hash of the previous one as way of achieving resistance against tampering. A consensus protocol assures the consistency of the ledger between the peers by agreeing on an order of the blocks. As a tool to identify and authorize network peers, blockchains usually make extensive use of public key cryptography.

Blockchain systems gained popularity through digital currencies ("cryptocurrencies") such as Bitcoin [6] and Ethereum [1], which application consist in the recording of money transfers between untrusted participants without the need of a central authority, e.g., a bank. Although being the most prominent one, possible use cases go far beyond financial applications. Other areas where blockchain applications are being deployed comprise for example supply chain management, medical health records or political voting [7].

2.1.1 Permissioned Blockchain Systems

Technologies like Bitcoin utilize a **public** blockchain, allowing everyone to join the network and participate in an anonymous manner. Public ledgers rely on consensus protocols like proof of work, involving economical incentives through computationally hard algorithms to avoid tampering with data. On the other hand, there exists the notion of a **permissioned** or **private** blockchain, in which participation is controlled by an authorization. Permissions might be given for different aspects of interaction and granularity. As an example, they might allow to control which peers join the network, read data from the ledger or write to it. While in a public network any peer might potentially be malicious, private blockchains are based on the assumption that peers pursue a common goal, but don't fully trust each other [3]. Since there is a control over the network participation, consensus can be established by using byzantine fault tolerant (BFT) protocols. The need to rely on a permission-controlling authorization comes with a cost of a partial centralization of the network. Permissioned blockchain can be used, e.g, in a scenario where two companies trade a certain type of assets with each other. While both companies trust their own peers and pursue the common goal of trading and, they might not have full trust towards the members of the other company. This problem is solved by establishing a set of rules, together with a common interface for the interaction between the two parties.

2.1.2 Smart Contracts

Since their first appearance in the field of financial transactions, blockchain systems have evolved over the years. Bitcoin and similar technologies provide a way to securely transfer cryptocurrencies between anonymous identities. While there is the possibility to integrate auxiliary data into Bitcoin transactions, the length is limited and might only serve to encode little pieces of information, as a kind of asset that is exchanged. The same limited behavior applies to a very simple scripts that can be included in transactions. Due to its rigid behavior and use case, Bitcoin is also known as a first-generation blockchain. Platforms as Ethereum [1] consider themselves as second-generation blockchains, offering the deployment of programs, so called smart contracts, on top of the ledger. These programs can contain business logic, triggers or conditions and can be seen as a trusted, distributed applications [7] [3]. Hand in hand with the cryptocurrency Ether, smart contracts in Ethereum can express logic such as an escrow that is holding funds until a set of conditions is met. The pieces of code are written in a Turing-complete language called Solidity and their deployment implies a certain cost in Ether, depending on the complexity of realized functionality.

2.2 Hyperledger Fabric

After explaining the basic concepts of this work in the previous section, we now look at the concrete implementation of a private blockchain system, which we focus on during this work. Fabric is a platform for permissioned distributed ledgers designed for the usage in enterprise contexts. The project is open source and released as part of the umbrella project Hyperledger, which aims at creating cross-industry blockchain technologies and was started by the Linux Foundation at the beginning of 2016 [8].

Fabric aims at fitting to a lot of different use cases throughout industry, therefore providing a modular architecture and broad range of configuration options. Another characteristic is the possibility to develop and deploy smart contracts in general purpose programming languages, namely Golang, Java and Node.js, instead of domain-specific languages as Solidity in Ethereum [1]. In the context of Fabric, smart contracts are called "chaincodes".

Other special characteristics of Fabric are as follows [9]:

- An execute-order-validate transaction processing, that favors scalability and performance (see 2.2.2)
- A pluggable consensus protocol for individual use cases and its trust models
- A native cryptocurrency is neither implemented nor needed for purposes of incentive-based consensus or smart contract execution

The following sections go more into detail about the Fabric platform. At first we will explain the architecture, describing the modules of the systems. The second subsection is about the architecture and flow of transactions, comprising the phases of endorsement, ordering and validation.

2.2.1 Architectural Overview

A Fabric network consists of a set of nodes, which are identified by a modular component called Membership Service Provider (MSP). Each node is part of an organization, which is a collection of nodes that trust each other. Typically, this group resides inside a real organization, or generally, a common host. That allows to define scenarios where several organizations participate in the same network, each with a certain number of nodes. As an example, two organizations can establish a network where they trade some kind of goods, while not fully trusting each other. Fabric defines three different roles that a network node can take [3].

The first role is called "peer". Each peer node maintains a local copy of the ledger, the ordered sequence of transactions, as well as the global state database. The latter is a versioned key-value store tracking the current state of the ledger and can be seen as a

supporting data structure, keeping track of the current state after the application of all valid transactions in the ledger. One subset of peers is called "endorsing peers" or "endorsers". These particular peer nodes simulate the outcome of proposed transactions on the chaincode. This process will be further explained in Section 2.2.2.

Secondly, a node can have the role of an *ordering service node* or shortly, *orderer*. Together with the other orderers in a network it forms the *ordering service*, which is responsible of determining a total order of transactions, being unaware of the application state. Orderer nodes collectively run a consensus protocol to agree on that order.

Clients, as the third and last role, issue transaction proposals and pass transaction to orderers for ordering. They are contributing to updates of the ledger and generally the state of the application.

A chaincode that is deployed on the network is associated to a so called "channel". There can be multiple channels running in a network, each connected to the ordering service. A peer node, on the other hand, isn't necessarily part of every channel, allowing different groups of peers to interact on different chaincodes [10].

2.2.2 Transaction Flow

This section explains the transaction flow, i.e., the process between the submission of a transaction by a client to its eventual commit to the ledger of every peer in the channel. From a high-level perspective that is the sequence of the three steps of execution, ordering and validation/commit, which is detailed and illustrated in the following.

Execution/Endorsement

During the first phase, called endorsement or execution phase, a transaction proposal by a client is simulated on one or more endorsing peers. Each of them performs initial checks on the wellformedness of the proposal, validates the issuers signature and verifies the authorization of the operation on the channel. Afterwards, the proposal, which contains both a function and input parameters, is executed inside of a Docker container corresponding to the chaincode on each peer. With the help of the state database, each endorser produces the transaction result next to a read-write set of the changes to the ledger state. Together with a signature of the endorser, this data is sent back to the client as a "transaction proposal response". The described process can be seen in Figure 2.1.

An so called *endorsement policy*, established at the moment of chaincode deployment by the system administrator, defines a set of endorsing peers necessary for the transaction to eventually get accepted after the execution phase. Endorsement policies are simply represented in the form of logical expressions on sets of peers, with semantics such as "demands a signature of one peer of each organization" or "demands peer signatures from two out of three organizations".

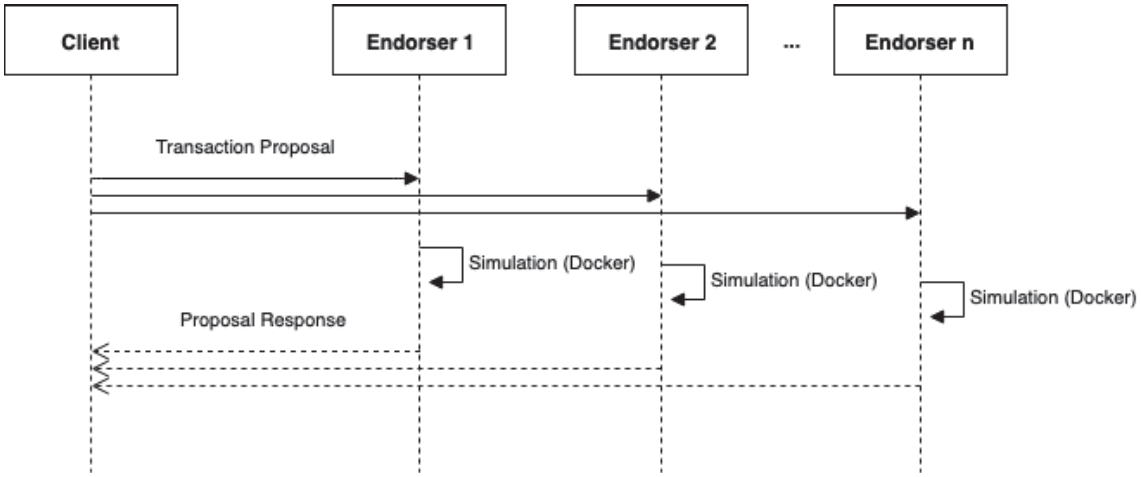

Figure 2.1: Sequence Diagram of Endorsement Phase

Fabric's design decision to simulate transactions before they enter the consensus step is in contrast to other blockchain systems and motivated by three main reasons [3]:

- The deterministic behavior of the operation is crucial for the consistency of the ledger among all nodes and has to be assured by performing various executions
- Running chaincode individually on only a subset of nodes favors performance by allowing parallel simulation of several proposals
- For reasons of confidentiality, the access of data related to smart contract execution may only be granted to a trusted subset of peers

After the client receives proposal responses from enough endorsers to fulfill the endorsement policy, and the read-write sets of each response are identical, the transaction can be created to be sent for ordering.

Ordering

A schema for the ordering phase can be seen in Figure 2.2. Once the transaction containing all endorser signatures is assembled by the client, it is sent to the ordering service. The task of the latter is to bring incoming transactions into a global order. This is achieved by running a consensus algorithm between ordering nodes. The actual content of incoming transactions is therefore not looked at, separating consensus completely from chaincode execution and validation [3]. The output of the ordering service are batches of transactions, called blocks, which are arranged in a hash-chained sequence. The grouping into blocks is done to amortize costs of network communication and updates on the ledger. The ordering service "cuts" a block, i.e., groups the

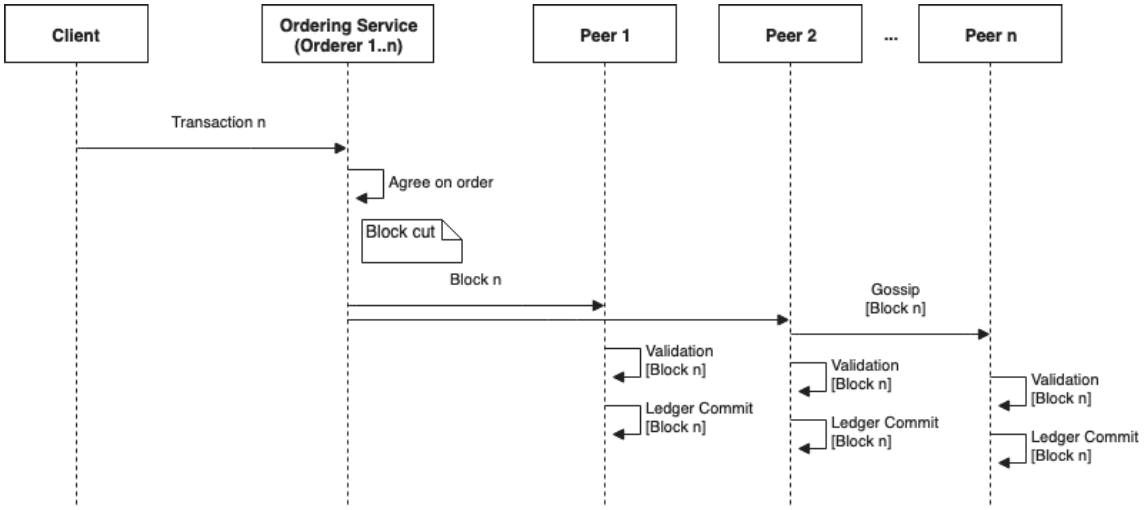

Figure 2.2: Sequence Diagram of Ordering Phase

current list of gathered and ordered transactions into a block, if one of the three following conditions are met:

- A configured block size in number of transactions is met. By default these are 10 transactions.
- The data of all outstanding transactions exceeds a configured maximum size.
- A timeout occurs before a new transaction arrives.

After a new block is cut, it is distributed among all peers in the network and enters the validation phase. This process might happen directly from the ordering service to a peer or among peers via gossip. The latter can be configured for networks where peers significantly exceed the number of orderers.

At the current moment, Fabric supports three different implementations of the orderer service, which handle consensus in a different way. The solo orderer assumes that only a single node is part of the ordering service. As a consequence it arranges transactions in the order they arrive and doesn't provide any crash-tolerance properties. That's why its usage is restricted to testing environments and not in production. For that matter, Fabric offers implementations based on Kafka [11] and since the recent release 1.4.1, the Raft protocol. Both use a leader-follower scheme where decisions made by a leader node are replicated to follower nodes.

Validation/Commit

After a block arrives at the peer either directly through the orderer or gossip, it individually validates its contents and finally adds it to its local version of the ledger. Every

transaction inside the block is first validated with regard to the endorsement policy. If this criteria is not met, because, e.g., the transaction was tampered with after endorsement, it is marked as invalid. During the subsequent stage, the serializability of transactions is checked. Block transactions are traversed sequentially, verifying that each read set doesn't contain a version of a value that was written in a preceding transaction. This can be achieved by comparing the versions of each key in the read-set with their current one in the peer's ledger for equality. Transactions that don't fulfill this criteria are also marked as invalid ones. After passing these steps, the local ledger of the peer is updated by appending the new block together with a bit mask which declares the validity of its transactions. Additionally, the write-sets of all valid transactions are applied to the peers state database.

2.2.3 Example

This section illustrates the transaction flow in a high-level example scenario, including the three phases explained before. The network in this example is going to contain a total of 6 nodes, 4 of which are peer nodes and 2 Raft-orderers. The peers are divided into two organizations with two peers in each, all acting as endorsers. For purposes of simplicity, these organizations are called Org1 and Org2. The chaincode that is running on the network's channel is based on a simple transfer of funds.

Each transaction specifies the balance of the two user accounts between which the transfer is performed. That means that if, e.g., User A transfers a specific amount of money to User B, this transaction will record the latest balance of User A in the ledger minus the transferred sum and correspondingly the latest state of User B's account plus the newly retrieved amount. A corresponding chaincode would consequently define a function that takes as input the identifiers of sender and receiver, together with the amount of transferred money. A corresponding pseudocode can be seen in Listing 2.1.

Listing 2.1: Transfer Function of the Example Chaincode

```
function transfer(sender_id, receiver_id, amount) {
        balance_sender = ledger.get_state(sender_id)
        balance_receiver = ledger.get_state(receiver_id)
        ledger.set_state(sender_id, balance_sender - amount)
        ledger.set_state(receiver_id, balance_receiver + amount)
}
```

The state inside the ledger is simply represented by a key/value store from which the shown code, as in Fabric's chaincode API, can conveniently read the current state of a key (*ledger.get_state*) and write a new state (*ledger.set_state*). In this case the keys are identifiers of the user accounts and the values the corresponding account balances. Additionally to the shown function, another one is needed to initially distribute all available funds in the network. Before transactions can be submitted to a channel,

the chaincode at first has to be installed to all endorsing peers, which are all 4 peers in this example. Afterwards, a client instantiates the chaincode, calling the initialization function which gives every peer a starting amount of 100 funds. The client can be on a separate machine in the network, but also reside on of the peer machines. The chaincode instantiation is additionally used to specify the endorsement policy that will be applied against the respective chaincode. In this example we assume that each one member of Org1 and one of Org2 have to successfully endorse in order for the transaction to eventually be accepted. In the following, the transaction flow of an actual transfer-transaction is described.

At first, the client creates a transaction proposal calling the transfer function, where *account_a* transfers the amount of 50 funds to *account_b*. The proposal, containing function name and arguments is then sent out to one endorsing peer per organization, since this is the minimum set needed to fulfill the endorsement policy. Both endorsers simulate the execution of the operation on the current application state inside a Docker container and eventually accept the proposal. They send back a proposal response where each added its signature and the determined read-write set. The read set contains the key/value pairs (account_a_v1, 100), (account_b_v1, 100). Since we assume that it is the first transaction, both keys have a version identifier of 1. The write-set contains the two tuples (account_a_v2, 50) and (account_b_v2, 150) where the version numbers are incremented and funds are updated according to the transfer function.

After receiving back both proposal responses, the client creates a transaction and sends it over to the ordering service cluster via Orderer 1. Once arrived, the Raft protocol lets both orderers agree on the ordering of the transaction between the two ordering nodes. We assume that there are further transactions arriving which eventually lead to a block cut, i.e., the bundling of transaction into a block that is distributed among all peer nodes. All orderer nodes are also appending the block to their local version of the ledger. Peer 1 and Peer 3 are the gossip leaders in their respective organization thus they receive the blocks directly from each of the orderers. The gossip protocol is then responsible for handing over the block to the remaining two peers. At last, peers will validate the blocks by verifying that each transaction has the right endorser signatures to comply with the endorsement policy and that transactions are serializable. Afterwards, the block is appended to each peer's local copy of the ledger and the transaction is completely processed.

2.3 Related Work

The performance characteristics of Hyperledger Fabric are analyzed in a work called Blockbench [12]. The authors propose a benchmarking framework for private blockchains, which consists in a number of different macro and micro-benchmarks. Next to Fabric, they measure the performance of two other mayor platforms, namely Ethereum and Parity. Fabric shows higher throughput, lower latency and disk usage throughout most

of the workloads, but the authors conclude that the generally low performance of these systems make them unsuitable for data processing applications of large scale.

There have been similar efforts to create an improved version of Fabric. A paper from Sharma et al [10] discussed how common practices of distributed databases can be integrated into the platform. Their changes mainly aim at increasing the goodput, as the number of valid transactions committed to the ledger, per second. As a first mechanism, they replace the arbitrary ordering of transactions with a reordering mechanism inside the orderer, arranging transactions into an order that reduces read-write conflicts. Secondly, they propose the early abortion of transaction that are prone to fail due to a version conflict inside their read-write sets. As a result of these changes, the authors reach up to a 3x higher goodput of the system. From a critical perspective, design of Fabric is changed in a sense that the ordering service is now looking at and working on the transaction contents, violating the original separation of responsibilities, as it was proposed in Fabric.

Instead of goodput, other work focused on decreasing the latency of Fabric. StreamChain [13] proposes a new processing model, where transaction are treated as a stream. StreamChain forms the basis for our work and is explained in detail in Chapter 4.

FastFabric [14] is a work that is mainly concerned about throughput. The authors propose various improvements, both inside of peer and orderer. On the orderer side, these include the separation of the transaction header from the payload for a more efficient communication inside the ordering cluster, as well as an internal transaction pipelining. As for the peer, proposals are the parallelization of the validation phase combined with a mechanism for block caching, next to extensive architectural changes. The latter include a separate cluster for block storage, and physically breaking up the two roles of peer and endorser in order to decrease the load on an individual peer machine. The diverse and mainly independent changes proposed in FastFabric made them good candidates to be migrated to our own improved version of Fabric. In particular, we make use of the proposed block caching, as we describe in Chapter 4.

Chapter 3

Fabric Performance Analysis

In this chapter, look at the performance characteristics of the Hyperledger Fabric platform. We do that by benchmarking and analyzing the different stages explained in the previous chapter, with the goal of identifying bottlenecks.

The benchmarks were performed in a small cluster of eight server machines, equipped with Intel Xeon E-2186G CPUs (12x3,80GHz) and 10 Gbps networking. The role distribution of the system is as follows:

- 5 peers, all endorsing
- 3 orderers in Raft mode

All of the peers are part of a single organization and the communication between peers and orderer is secured by TLS.

3.1 Workload Generation

The chaincode used for the performance evaluation represents the functionality of a simple key/value store. The benchmarks utilize the two functions "insert" and "read", which respectively set and get a value, to and from the ledger. The chaincode functionality is kept as simple as possible to not significantly impact performance from the first stage on (endorsement).

A slightly extended version of Fabric allows us to run a workload, i.e., to perform a predefined list of operations through Fabric's client CLI. Since the latter only allows to perform a single operation at a time, it otherwise would be necessary to call it separately for every operation, adding a significant overhead through newly established network connections on every call. Our modified version directly integrates parallelism into the workload processing, allowing the execution of sets of operations on a specified amount of threads. We can look at these threads as parallel clients issuing operations.

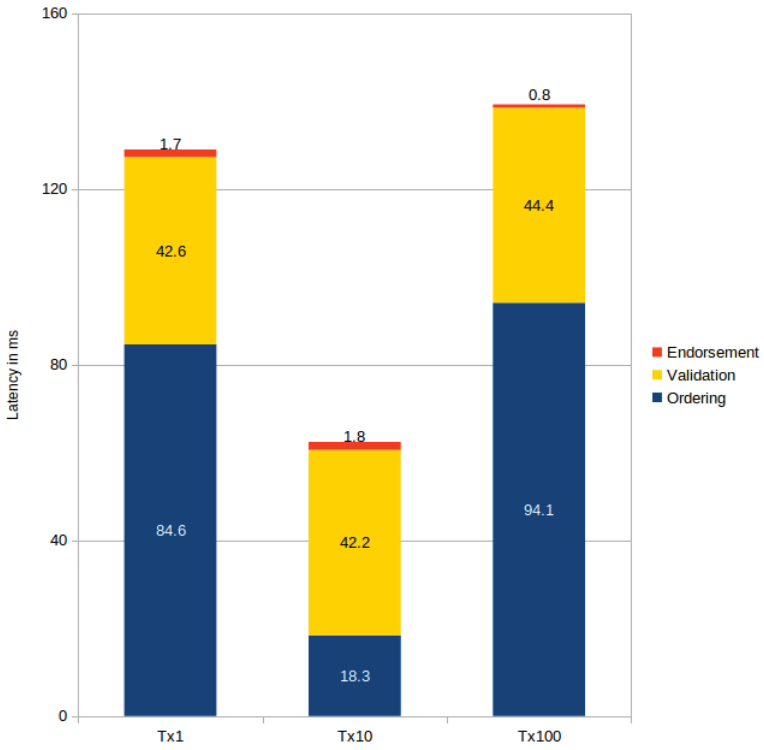

Figure 3.1: Breakdown of Latency in Fabric

There are different workloads used to retrieve the results presented in this chapter. They all contain a total of 10.000 operations and differ in the distribution of read to insert-operations. However, we will mainly focus on a single workload, which has a ratio of 10% reads to 90% inserts, where all inserts add completely new values instead of updating already present ones. This is due to the fact that we don't want to introduce failed reads in the validation phase, which might potentially influence latency results at that point. In order to assure that read values are present in the ledger, a loading phase is performed before the actual workload, inserting 10.000 values from the read data set. All values included in the workload are arbitrary strings of 1KB size.

A measurement of the latency of the system by the 3 different phases of Endorsement, Ordering and Validation with 1 thread and the explained workload shows the distribution of the average time spent per transaction in the system. Also, three different block sizes of 1, 10 and 100 Transactions/Block are compared to see if latency is influenced. In the following, block sizes will be abbreviated in the form of Tx*n*, where n is the number of transactions in a block (e.g., Tx10 for blocks of 10 transactions). The result can be seen in Figure 3.1.

In case of single transaction blocks, ordering latency accounts for a significant percentage of total time, claiming about two-third of it. The rest is mainly dominated by the validation phase and a little part by endorsement. Ordering time significantly decreases for Tx10, which is the default configuration of Fabric and shows a good tradeoff

of the batching characteristics of reduced disk flushing time and additional overhead by "filling up" a block. However, it is to be expected that the ordering time for Tx10 is equal or even bigger than Tx1 in a scenario, where transactions are issued very infrequently and block cuts might happen even after timeouts. Tx100 shows a latency in the similar order of Tx1 again, as disk flushes might be efficiently amortized but every transaction spends a much longer time inside the ordering service, before a block cut happens. The time of the validation phase is showing a constant character through different block sizes.

Validation and endorsement results are not significantly influenced by the block size. Endorsement happens on a per-transaction basis and is therefore not effected by design. The range of values from 1,8 ms in Tx10 to 0,8 ms in Tx100 can be explained by variances in execution time. The validation phase is responsible for 43 ms, which in the default case of Tx10 makes around two-third of the total. The reason can be found in expensive disk flushes happening at the end of the phase before transactions are finally part of the ledger.

In the following sections we will take a look into the performance characteristics of the three stages, looking mainly at throughput and reason about the causes of the observed results.

3.2 Endorsement Phase Analysis

In this section we are looking at the throughput of endorsement, as the number of processed, or outgoing operations of that phase, per second.

In order to measure endorsement throughput, it was necessary to manipulate the behavior of Fabric's CLI client. When measuring endorsement under normal circumstances, it reflects the throughput of the ordering service. The reason for that is that the client blocks from the moment it sends a transaction to the ordering service, waiting for its response until it continues to process the next transaction in the workload. The ordering service has to initiate the ordering of every transaction before sending out a response — a process involving disk writes and other steps related to the processing of previous transactions. As a result, endorsement and ordering measured together form a closed loop. We deactivated the blocking wait for the mentioned ordering service response inside the client. That measure was performed uniquely for the purpose of obtaining reference numbers, since it essentially breaks the correct functioning of the orderer and can't be performed in an integrated scenario with ordering and validation.

Through the measurements in this chapter, we define the endorsement policy so that only one successful endorsement is sufficient. This implies that every proposal has to be sent to only one endorser and that, as our workload is equally distributed among the 5 endorsers, each of them performs 2000 endorsements. The results can be seen in Figure 3.2, which shows the throughput depending on increasing load on the system by parallel clients. Since the endorsement is done on a transaction level, the

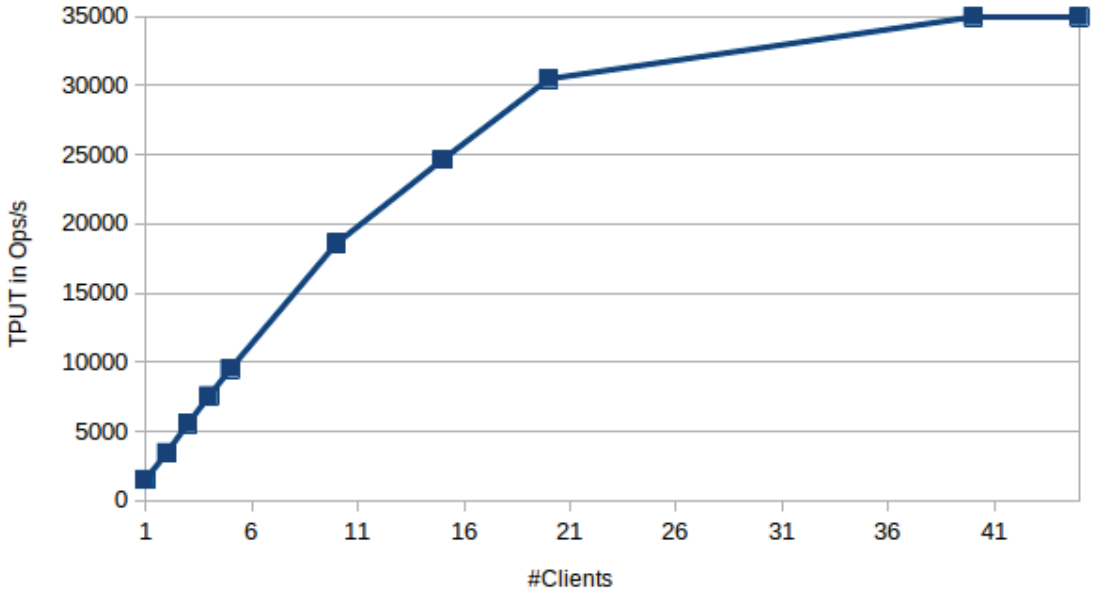

Figure 3.2: Throughput of Endorsement Phase

block size is not relevant at this point.

The 5 endorsers in the network are able to process 1500 Ops/s coming from a single client. Afterwards, the numbers increase proportionally to the number of clients, until the latter match the number of endorsers at a throughput of almost 10.000 Ops/s. The slope decreases and finally reaches a maximum of 35.000 Ops for 40 clients.

3.3 Ordering Phase Analysis

Our ordering service consists of a total of three orderers and is configured to run in Raft-mode. The results of throughput measurements at the ordering service as the second step of the processing pipeline of Fabric can be seen in Figure 3.3.

Next to the high latency caused by disk writes on the orderer, we see a low throughput of around 58 Ops/s for the case of Tx1. It stays constant for Tx10, being in fact around 10x higher at around 560 Ops/s. In case of Tx100, disk-writes are further amortized, leading to a significantly higher throughput with increasing load. At 20 clients a maximum of 4500 Ops/s is reached. A similar increasing trend can be expected when increasing the block size further. The amortization of disk-writes by blocks comes with the cost of high latency, as seen in [figure: Latency breakdown]. In the following, we want to take a look at the latency if expensive disk writes are taken out of the picture and disable them by letting the orderer operate on an in-memory ramdisk instead of hard-drive. The usage of volatile storage instead of the hard-drive can have reliability implications in the system, as the failure of a node leads to data getting lost. These

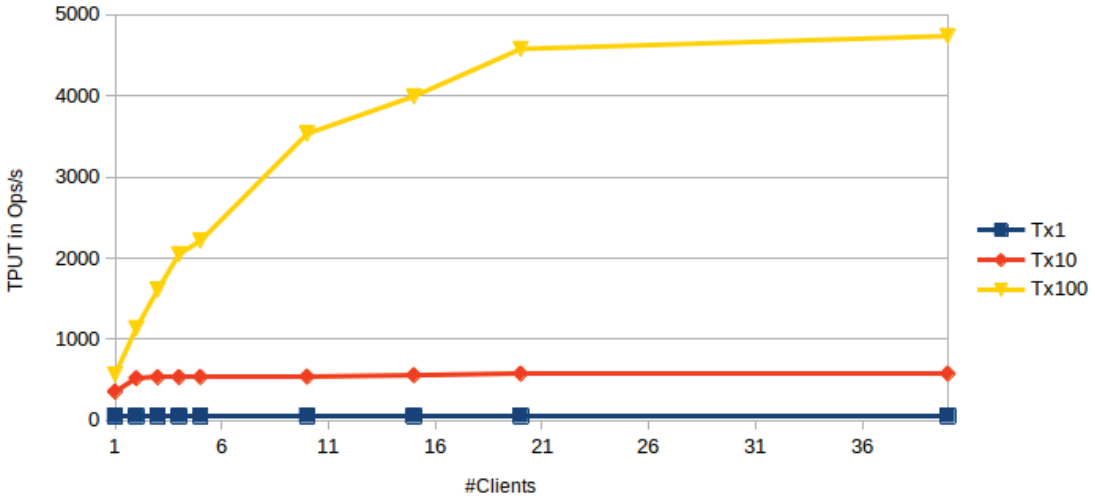

Figure 3.3: Throughput of Ordering Phase

changes are therefore not applicable in a production system. The resulting graphs can be seen in Figure 3.4 and Figure 3.5.

If the cost of disk writes is almost negligible, there is no advantage of batching transactions. This is represented in an almost proportional increase of latency with number of transactions in a block. Single transaction blocks, which usually imply a high number of disk flushes now show a very low latency of less than a millisecond, while transactions in "big" blocks of 100 transactions stay on average almost 90 ms inside the ordering service. Regarding throughput, blocks of 10 transactions benefit the most, offering the highest efficiency per single transaction and reaching more than 12.000 Ops/s. Endorsement throughput as shown in Figure 3.2 is still higher at any load and not limiting throughput.

3.4 Validation Phase Analysis

During the validation phase, the processing of a block can be divided into three different stages, namely:

- VSCC Validation for each of the block transactions (VSCC)
- Serializability check of Read/Write Set among block transactions (RW Check)
- Applying the blocks state changes in LevelDB and append block to ledger copy on disk (Commit)

The latency breakdown in Figure 3.6 shows the distribution of spent time among these three steps, again comparing it between three different block sizes and 1 client.

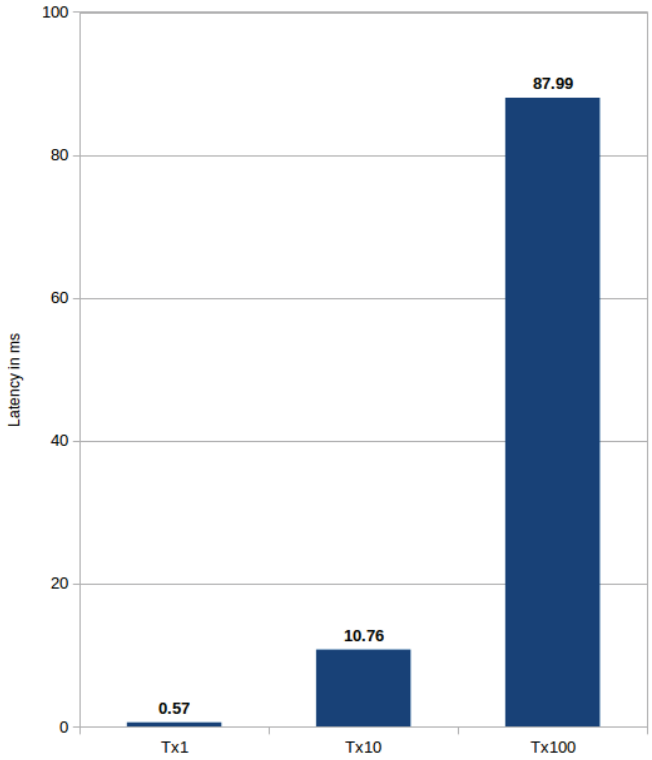

Figure 3.4: Latency of Ordering Phase (Orderers in RAM)

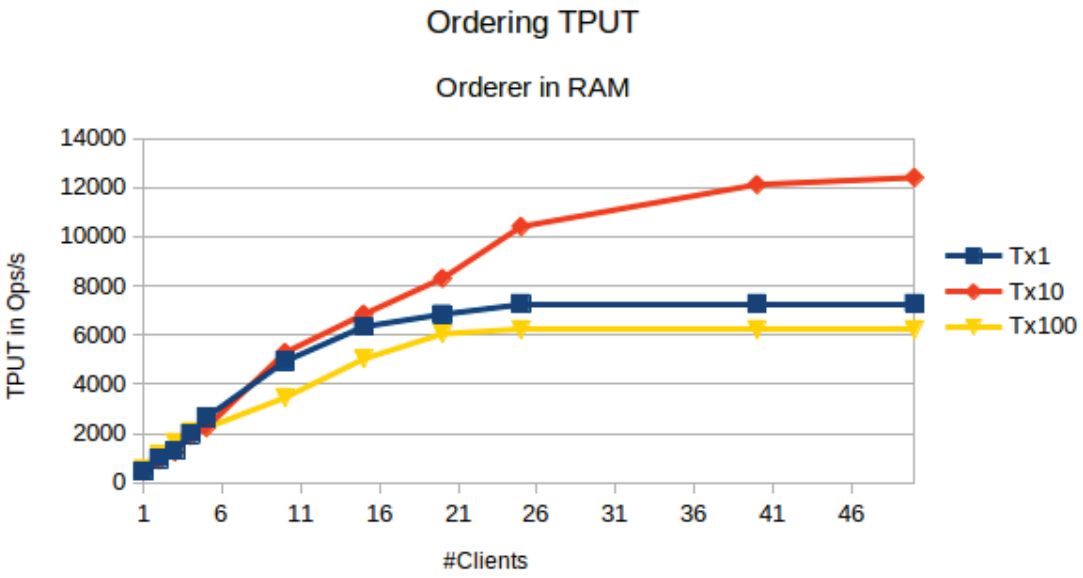

Figure 3.5: Throughput of Ordering Phase (Orderers in RAM)

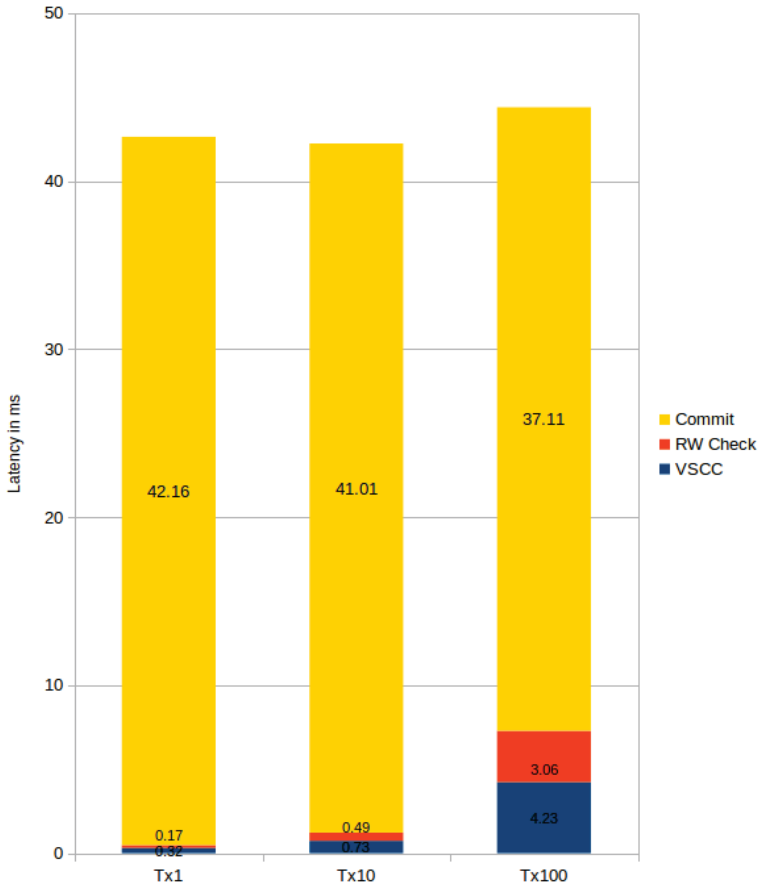

Figure 3.6: Latency Breakdown of Validation Phase

In order to remove unnecessary overhead in the third step, we disabled Fabric's History DB features, which, next to the current ledger state, saves the history of state changes to enable state queries at a certain point in time and uses a LevelDB store on disk. Since it is a rather rare use-case, we are not interested in performing state queries, nor measure their performance implications.

For all three block sizes, the commit step claims the big majority of the spent time, taking 42 ms and 99% in case of Tx1, and slightly decreasing to 37 ms and 89% for Tx100. The high latency is explained by the disk writes involved in updating ledger and state on the peer. The reduction can be explained with a larger amount of data being written to disk more efficiently. The first two steps of VSCC Validation and RW Set Check are operating on every individual transaction in a block thus increasing latency from a sum of 0,20 ms for Tx1 to 4,3 ms for Tx100, still spending the minority of time in validation.

The throughput of the validation phase, as seen in Figure 3.7 shows a linear trend for Tx1 and Tx10, being at 23 and 230 Ops/s respectively. Since there is no parallelism involved inside of validation, throughput is determined by the inverse of the commit

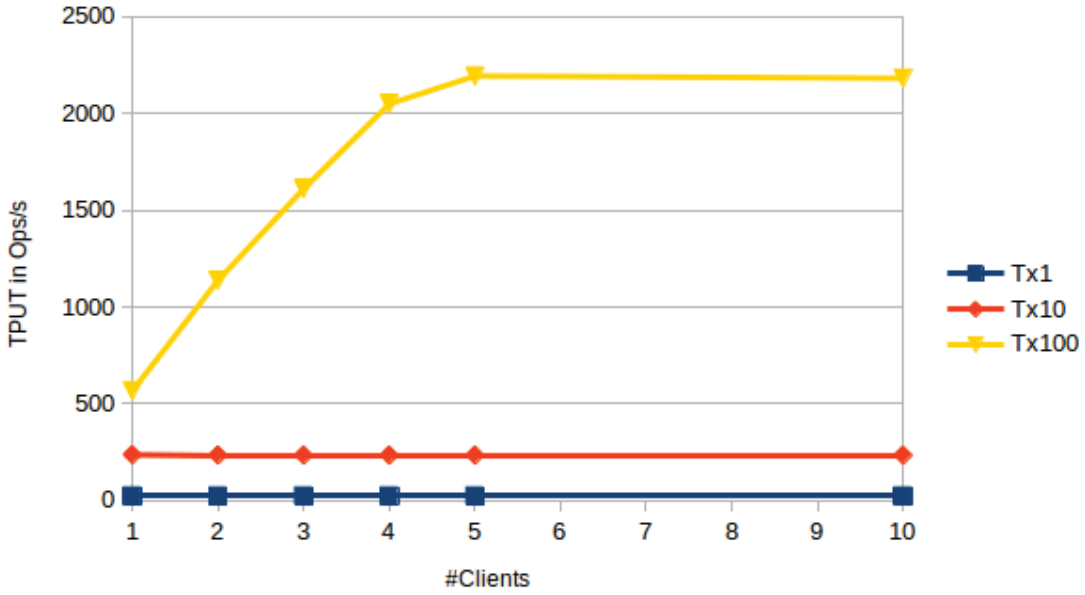

Figure 3.7: Throughput of Validation Phase

latency of a block. Being significantly lower than ordering throughput in these cases, the validation phase acts as a bottleneck in the whole system. For blocks of 100 transactions, ordering becomes the bottleneck up to a number of 5 clients, where validation eventually reaches its maximum of 2300 Ops/s.

Furthermore, we want to see how validation latency and throughput turn out if disk flushes as biggest source of latency are eliminated. For that purpose we put peer and orderer data into RAM. The corresponding graphs can be seen in Figure 3.8 and Figure 3.9.

The latency distribution shifts away from the commit step and towards VSCC and RW Check. The latency of Tx1 decreases to 0,6 ms thus showing the smallest latency. The overall time for Tx10 decreases to 0,2 ms and for Tx100 to almost 10 ms. As seen before, the first two steps increase their latency with growing block size. This time a significant growth in the third step can be observed as well. Correspondingly, a higher throughput is reached for all block sizes, going up to 7000 Ops/s for Tx100.

During validation, similar observations as for the ordering phase could be made. Increasing the block size increases the throughput effectively, since the processing of an individual transaction becomes more efficient. Likewise the time until an individual transaction ends up in the ledger is delayed. That becomes even more clear when letting the peer write to memory and latency is not strongly dominated by disk flushes.

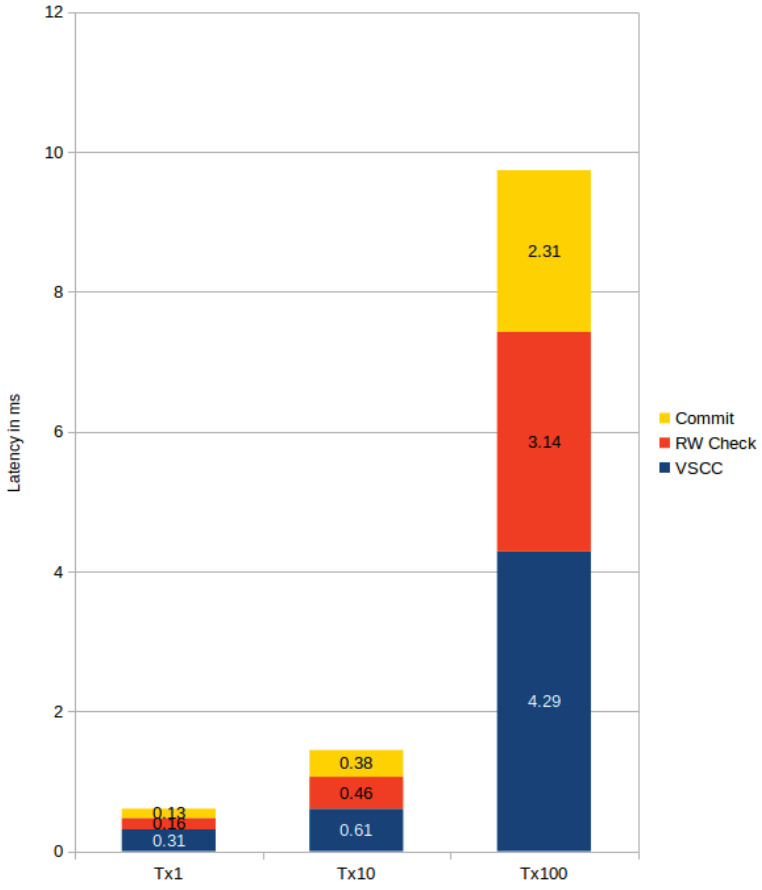

Figure 3.8: Latency of Validation Phase (Peers and Orderers RAM)

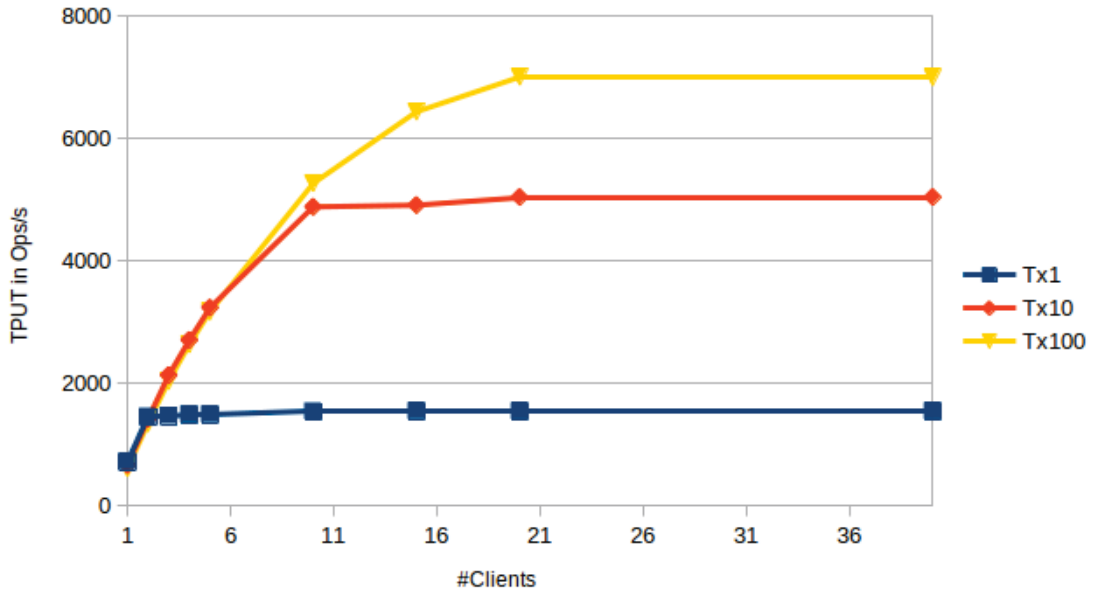

Figure 3.9: Throughput of Validation (Peers and Orderers RAM)

3.5 Conclusions

In this chapter we have taken a look into the performance characteristics, namely latency and throughput numbers, for the three phases that form part of Fabric's pipeline. By default, Fabric batches transactions into blocks to amortize the expensive costs of disk writes and achieve high throughput. Latency on the other hand gets compromised, because of the time that a single transaction passes inside the orderer and peer before being committed to the ledger. From the latency breakdown shown in Figure 3.1, it can be concluded that the default block size of 10 transactions indeed leads to the lowest latency. We also showed that, if disk flushes were negligible, a block with a single transaction results in an almost two orders of magnitude smaller latency in both ordering and validation. In Figure 3.10 we show the distribution of the overall spent time assuming free disk writes.

In case of Tx1, all three phases contribute equally, with validation and ordering time being in the order of endorsement. The bigger the blocks get, the more time does ordering claim. The graph shows that endorsement time goes down as well, when peer data is kept in-memory. Accesses to the state database, shared between the logic of endorser and peer, are getting cheaper.

Based on these results, in the following chapter we will explain how we can achieve both low latency and high throughput by processing transactions as a stream. At the same time we assure reliability of the system by adding back the hard-drive as the underlying storage of the peer, but doing so without reducing throughput.

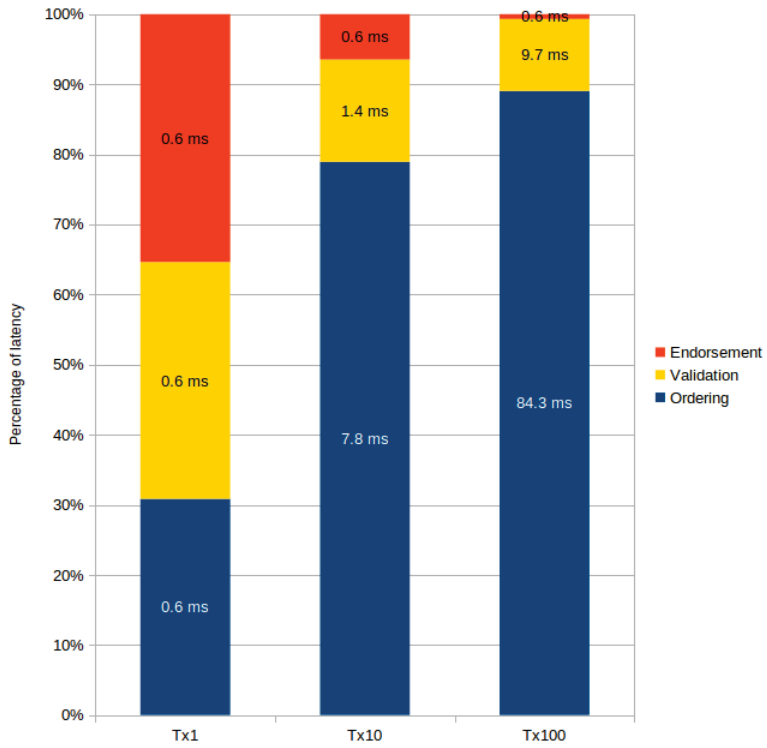

Figure 3.10: Latency Breakdown (Peers and Orderers RAM)

Chapter 4

Design and Implementation

In this chapter, we present the optimizations introduced into the Hyperledger Fabric platform as measures to reduce latency, while maintaining high throughput.

This work is mainly based on the StreamChain proof-of-concept (P.o.C.) [13], which proposes a new design for distributed ledgers in which transactions are processed in a streaming fashion, rather than grouping them into blocks. We implement StreamChain for the latest version of Fabric as well as explore and implement improvements to further increase performance of the system.

StreamChain proposes the update of state and ledger at defined intervals to amortize disk writes. That corresponds to what was originally achieved by grouping transactions into blocks. As a proof-of-concept, the underlying storage is replaced by an in-memory store, which simulates amortized writes. In this work, we implement real batching of transactions, so that we take a step towards a real production system, where data is written on persistent storage. As the biggest contribution, we implement a batching mechanism inside the commit step of the peer, defined on transaction level and configurable from the outside. In the remainder of this work, we will refer to this configurable size of the batching as the batching parameter n. If backed by disk, the ordering service would form a bottleneck of the system. While it is not recommendable to keep the data of a solo-orderer in memory, as a failure of the node leads to ledger data getting lost, we assume that multiple orderers in Raft-mode offer sufficient replication to consider the non-persistent ledger storage as safe. As a matter of fact, the batching of disk writes introduced in the peer can be applied for the orderer nodes as well.

The work called FastFabric [14] proposes a number of changes to Fabric's architecture and internal processing, successfully increasing throughput at all stages of Fabric's transaction processing. For our implementation we consider one of these changes at the validation phase performed inside a peer node. Blocks received by the orderer are transferred in a byte format and their contents have to be frequently unmarshalled. A temporary cache avoids that this costly process is done several times. The remaining ideas proposed in the paper are either already covered by our implementation, or

imply significant changes in Fabrics architecture.

From a detailed analysis made for the validation phase, which forms the bottleneck for throughput of the system, we can derive further minor improvements. That includes the extension of the validation pipeline by an additional stage and the removal of a code path, which reads in a configuration parameter redundantly.

As the peer receives a blocks and before introducing it into the validation pipeline, the unmarshalling of block contents is creating a significant overhead. While this problem is already addressed by caching the block contents, it has to be done once, slowing validation down. The preparation is followed by the first pipeline step, the VSCC validation, which is already parallelized and doesn't show potential for optimization. The check of transaction signatures as cryptographic operations reveal themselves as a source of high latency, that cannot be trivially improved.

In the first section of this chapter, we explain the original design of the StreamChain P.o.C. and detail the implied changes in the architecture and functioning of Fabric. The second section addresses the implementation of batched writes as next step of porting the StreamChain ideas to Fabric. The block caching changes originating from FastFabric are explained in the third section and lastly we take a look at additional changes leading to our final implementation.

4.1 Porting the StreamChain P.o.C.

StreamChain revisits the idea of batching multiple transactions into blocks and proposes the processing of transactions in a streaming manner with the goal of decreasing end-to-end latency 4.1. Additionally to low latency, it comes up with the idea to keep disk writes at block boundaries to provides high throughput thus combining the fast processing of single transaction blocks with the advantage of amortizing writes that comes with batching them to blocks.

Streaming behavior is proposed in both the orderer and the peer. It is proposed that the ordering service sends out transactions immediately after consensus on their order is established instead of waiting for a block to accumulate. The orderer's signature included over the data of a block is sent out periodically at virtual block boundaries to peers, confirming the last number of transactions, which are then committed to the ledger of orderers. In the meantime, peers can validate single transactions in a streaming manner and stage the state changes to endorse and validate other transactions. Data is staged until a correct signature of the ordering service allows to finally commit it and a consistent ledger state is established. In case of receiving an invalid signature, the peer simply rolls back the staged data. This is a safe operation, since previous endorsement results depending on the discarded stage changes will eventually fail in the validation phase. In the original Fabric, the state DB as well as the ledger are updated at batches in form of blocks. As a consequence of the afore-mentioned concept, Streamchain implements state updates at the granularity of transactions, while updating the

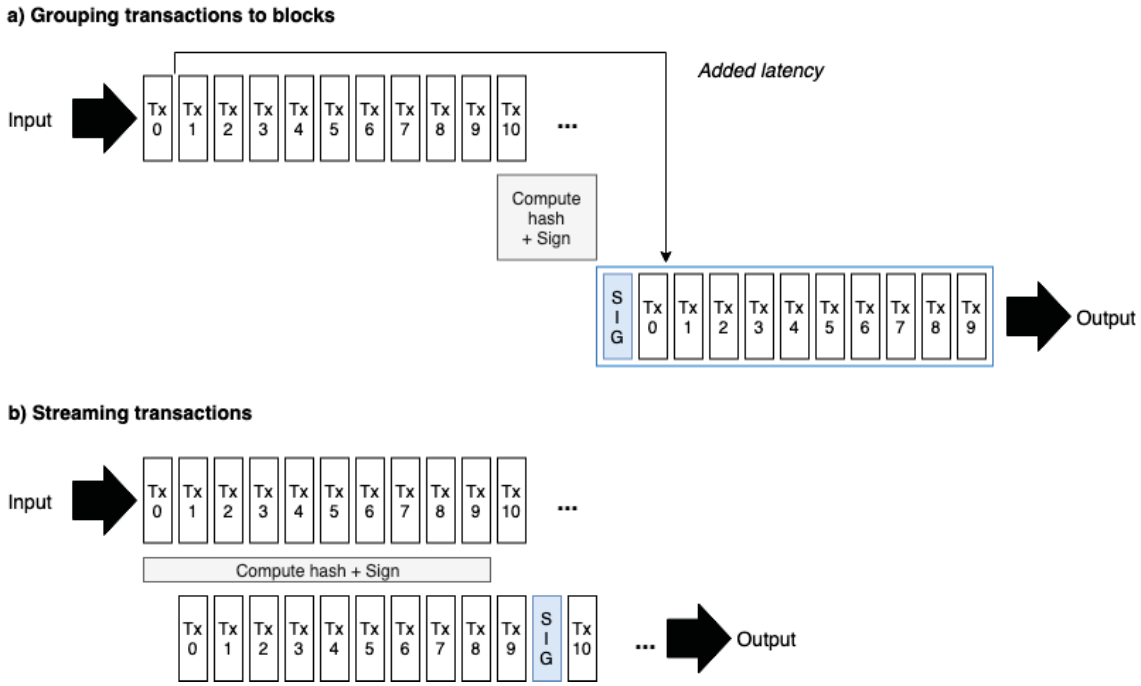

Figure 4.1: Conventional Transaction Processing in Blocks vs. Stream Processing

ledger at block boundaries.

In this section we explain the original implementation of StreamChain as a proof-of-work in Fabric 1.1. The ordering service is configured to use blocks of a single transaction, as seen before in the previous chapter. StreamChain discards the notion of blocks and merely sees them as envelopes around transactions, which doesn't contain any block signature. Ledger commits at batches of blocks at the orderer are simulated by using a ramdisk as a backing storage for the ledger. Furthermore, the code paths responsible for the creation and validation of ordering service signatures are completely removed in order to approximate the low latency of periodic signatures over transactions. At the peer, the low overhead of periodic disk flushes at state DB and ledger is also simulated by a backing ramdisk. Moreover, the logic of the validation phase was changed to process incoming streamed transactions more efficiently.

The validation phase inside the peer has been reorganized into a software pipeline of two stages (Figure 4.2). The first stage comprises the validation of transaction signature checking (VSCC) in parallel. While Fabric since version 1.1 performs the verification of transaction signatures in parallel on a block level, StreamChain implements the same mechanism to parallelize the signature verification across the (single-transaction) blocks. The second pipeline step contains the remaining logic of RW set check and state updates. As a result, multiple levels of parallelism in form of thread pool and pipelined execution are introduced to favor throughput. Finally, the state DB is modified so that it utilizes fine-grained locks instead of a global read-write lock. Since both endorsing and committing logic access the state, having multiple locks covering each

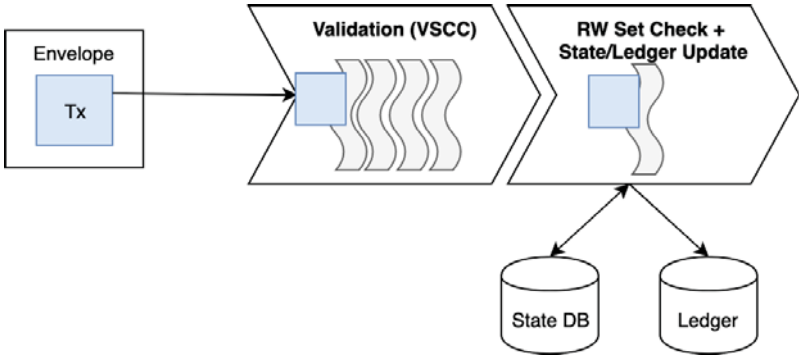

Figure 4.2: Pipelined Execution of Validation Phase

a subset of the key-space, decreases the impact of the frequent state updates at the granularity of transactions.

As a part of this work, we implement StreamChain for Fabric version 1.4.1 and port all of the modifications, including the validation pipeline, fine-grained locking and removed logic for ordering signatures. Moreover, we implement internal batching of transactions and assure that the resulting system is still reliable and compatible with Fabric.

4.2 Reducing Disk Writing Costs

As a first step, we look at the code path of the commit step and identify the parts where disk flushes are involved. That is especially interesting for the update of the current ledger state, which comprises the updates of various data stores on disk. In the following we show a list of the involved data stores. Afterwards, we explain each of them and how respective disk writes are avoided or batched, in more detail.

- The ledger: Copy of the complete chain of blocks and their transactions
- History DB: History of state changes allowing to query the state at a certain moment in time
- Private data store: Data records only visible to a subset of organizations
- Block indices store: Supporting data that helps to localize blocks and transactions in the ledger data, as well as certain metadata for each block for faster access
- State DB: Current state of the application

As mentioned in the previous chapter, the possibility of **history** queries shall not be considered in an evaluation of Fabric's performance thus we deactivate it in the configuration of the peer and don't take a further look at it. Also, our implementation keeps

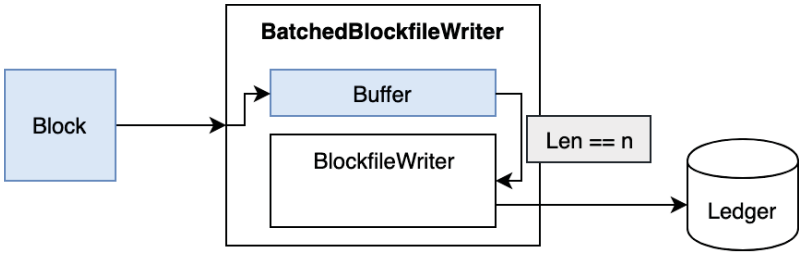

Figure 4.3: Data Flow through BatchedBlockfileWriter

the State DB in-memory by default and therefore doesn't make it subject to batching. One reason of keeping it in memory is the fact that it can be restored easily in case a peer node fails, the other one that there are current efforts as part of the next Fabric release, that move the state DB towards memory as well. Nonetheless, the techniques that we use to batch disk writes, described in the following, apply to the State DB as well.

The **ledger** is stored in a growing number of files on the disk and contains the contents of each block in a byte data format. A structure called *BlockfileWriter* is responsible for appending new block contents to the current file. Whenever the current file reaches a maximum size, internally defined as a constant, it opens a new file with growing index in its name (*blockfile_0, blockfile_1,..*). We wrap the BlockfileWriter in a construction called *BatchedBlockfileWriter*, which next to a reference to the former, holds an array of block data. More specifically, that array contains tuples of block data contents to be written, associated with the name of the file they are going to be written to. This array serves as a buffer of blocks (in our case equivalent to transactions), before writing them to disk. Once the append-operation of the underlying BlockfileWriter adds the n-th element to the array (n being the batching parameter), the buffer is emptied, writing each block's data to the associated file, where writes to the same file are grouped together before flushing them. That process can be seen in Figure 4.3. The default block file size defined inside Fabric and StreamChain is 64 MB, which means that flushes to multiple files during write-out only occur in very rare cases or extraordinarily high batch sizes, where they are amortized anyway. The BatchedFileWriter additionally keeps track of the length of the byte contents it's currently buffering, so that it can be determined if the current file exceeds the maximum file size, although parts of its data are still in the buffer.

On a higher level, every instance of BlockfileWriter is replaced by the BatchedFileWriter, offering the same interface. That has the advantage, that in case of a batch-size of 1, it can directly delegate function calls to its underlying BlockfileWriter, in which case it behaves like it would in original Fabric. On a higher level, whenever ledger contents on the disk are queried, the BatchedBlockfileWriter is forced to write out its buffer immediately, ensuring that the file ledger is in a consistent state. That happens in rare cases, e.g., when a specific block is queried by a Fabric client, or a range of blocks is synchronized after a peer node fails. A write-out is also enforced if a

certain timeout without incoming blocks is reached, in order to avoid that block data is being hold in memory for longer than necessary.

The **private data store** is, similarly to state and history DB, implemented through LevelDB. It persists data which is declared as private inside the chaincode, and allows data exchange between a subset of organizations. There are rather rare use-cases of private data, where multiple organizations participate in a blockchain network, but only some of them shall have access to a specific information. However, the private data store is flushing data to disk, even if private data is not made use of inside transactions. The reason lies in the fact, that it stores the number of every new block as a way of checkpointing. In order to integrate batching, we buffer these checkpoints for every new block, until we either reach block *n*, or process a block with actual private data. As an alternative approach, private data entries could be fully cached and only written out at intervals, which favors scenarios which make heavy use of private data.

The **block indices store** acts as a support for the ledger and State DB. It keeps a number of indices per block for faster access of data. Examples are the filename and byte offset of a block in the file-ledger, the block number that a specific transaction belongs to, or the validation status of every of a block's transactions. Since the indexed information is queried with every chaincode execution and validation inside the peer, we keep the indexed entries in a temporary map and utilize a counter to keep track of the number of blocks which indices are currently being cached. All functions which retrieve a value from the block index store first check if the related entry is being held in the cache, and otherwise proceed by querying it from disk.

After implementing the explained changes, the batching parameter n allows the amortization of all of the disk flushes happening in the previously shown list of data stores. After providing StreamChain with a "real batching" at the peer nodes, we can now look into changes, that further increase the throughput of SreamChain.

4.3 Reducing Deserialization Costs

Fabric uses gRPC [15] for the communication between participants of the network. It converts data into Protocol Buffers [16], a byte-based serialization format, before sending it out, and needs the receiving side to unmarshal the contents into the original data structure. In this case, we look at blocks being sent out from the ordering service to a peer node. The block data structure is designed to be layered, so that its contents can be marshaled or unmarshalled selectively and at different levels of granularity. Since parts of the block are used several times during the validation phase, and unmarshalled contents are not stored in a cache, the frequent memory allocations by unmarshal processes turn out to be costly. As a solution, we implement a cache at the block-level, which stores unmarshalled block contents to avoid the work of unmarshalling to be re-done.

The authors of FastFabric [14] demonstrate that unmarshalling is happening in

multiple places and is an expensive operation. Therefore they suggest a caching mechanism for block contents, which are frequently unmarshalled while a block passes the pipeline.

The caching is realized by extending the block data structure inside of Fabric. More precisely, we wrap the original block structure together with fields of its unmarshalled contents. In that manner we wrap all the layers of a block and also substitute the functions responsible for unmarshalling each layer. Our functions extend the original ones in a way that the layer to be unmarshalled is first looked up in the corresponding field of the higher-level layer. In that case, we can simply take these contents out of the cache and skip the unmarshalling. Otherwise we unmarshal once and set the field of the higher layer accordingly, caching the value. As an example, our new block data structure consists of a pointer to the original block data, as well as two arrays of cached envelopes (containing transactions) and cached metadata, respectively. Whenever a specific envelope of the block is unmarshalled for the first time, we do that passing over the original block data structure, since the result is not cached yet. Afterwards, the output of the function is stored in the array of envelopes of our modified block, from where it can be taken in following unmarshal requests. Each envelope again contains various layers of data, so that this procedure is repeated throughout all layers of the hierarchy.

Upon arrival of a block at the peer for validation, its contents will never be modified. As a consequence, cached data doesn't have to be secured against concurrent accesses from inside the validation pipeline. The worst case is that two threads want to obtain an uncached layer at the same time and both unmarshal it, leading to the same result. The block cache has the potential to decrease the latency throughout the whole validation phase, as well as further increase its throughput.

4.4 Further Optimizations

Finally, we look at how we further improved the performance of StreamChain's validation pipeline and achieved higher throughput. We perform an analysis to identify code paths which are responsible for significant latency in each pipeline stage and implement improvements. As a result, we extend the pipeline by an additional stage.

The processing and storage of blocks are still the causes of a majority of time spent in the second pipeline stage. These steps however, are directly depending on the batching parameter and we see no more potential to increase throughput beyond disk flush amortization. A significantly high effort is caused by a step involving private data, although it is not used inside our blocks. We identify the read-in of a configuration parameter that can be moved to the constructor of the higher-level structure to avoid that time being spent for every block. At the end of this pipeline stage, information about the current height of the ledger (being the new number of blocks in it) is updated inside the peer. That information is being held for the sake of synchronizing peer nodes

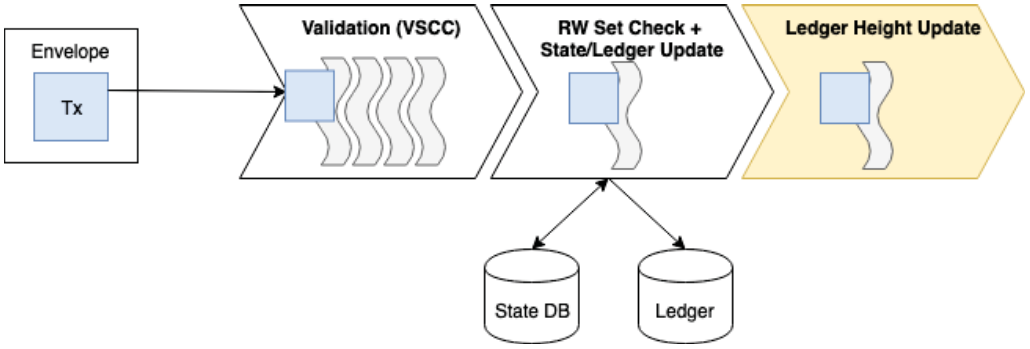

Figure 4.4: Extension of the Pipeline by a Third Stage

among each other, in the case that gossip is being used. While StreamChain does not make use of it, that functionality is kept for completeness. Internally, that translates to incrementing an atomic integer. Since this call has no dependent variables from the previously performed block commit, it was transferred into a third pipeline stage to increase throughput of the pipelined execution (see Figure 4.4).

4.5 Conclusions

In this chapter we explained how we implemented StreamChain for the latest version of Fabric, integrated real batching as in a production system, and enriched it with block batching. Beyond that, we identified two improvements in the validation phase, which benefit both throughput and latency.

In the next chapter, we perform a series of measurements for the introduced changes described in this chapter, and show by how much performance is improved in comparison with the original implementation of Fabric.

Chapter 5

Evaluation

After explaining how we implemented the ideas of StreamChain in the previous chapter, we perform benchmarks to evaluate the performance in comparison with the original implementation of Fabric.

For the collection of results, we use the same setup as in Fabric's performance analysis in chapter 3. We operate a small cluster of eight servers with Intel Xeon E-2186G CPUs (12x3,80GHz) and 10 Gbps networking. The blockchain network is composed of 5 endorsing peers and a Raft ordering service of 3 ordering nodes. A workload with a read-update ratio of 10/90 and 10.000 operations is run on a previously loaded dataset of the same size.

StreamChain proposes the pipelined execution of the block validation in the peer node. This measure aims at increasing the throughput at this stage, which forms the bottleneck in the system of Fabric. We implement 3 pipeline stages, VSCC Validation, RW set check/ledger commit and ledger height update, with the first one utilizing parallelism at a block level. In the first part of this chapter, we compare throughput numbers, first taking into account different parameterization of our pipeline implementation in the validation phase. Afterwards, we look at how throughput relates to different sizes of our internal batching mechanism at the peer. In the second section we focus on the comparison of latency for both versions of Fabric by examining the relationship between latency and throughput and compare latency breakdowns of the different systems. At the end of this chapter, we examine a modified version of the throughput, where only valid transactions are considered. We refer to this measure as the goodput of the system.

5.1 Throughput

In the following, we measure the throughput of the validation phase in our StreamChain implementation (being the throughput of the overall system), increasing the number of threads in the VSCC stage. The result can be seen in Figure 5.1, where we

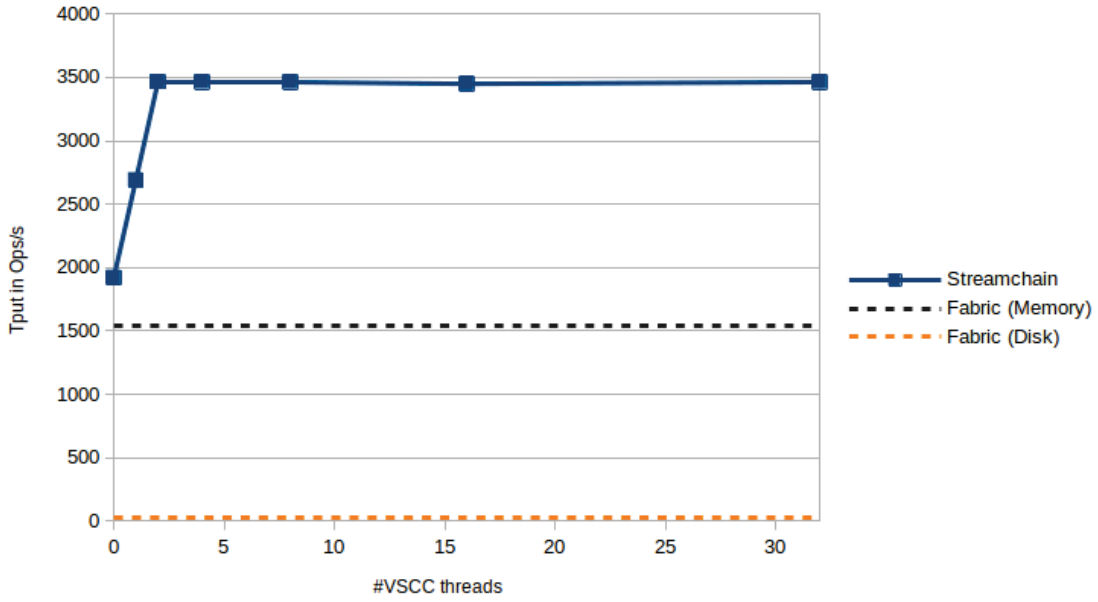

Figure 5.1: Throughput with increasing VSCC Threads in Validation Pipeline

add two baselines, respectively for the throughput of original Fabric on disk as well as backed by RAM. To create a comparable scenario, where latency is lowest, we configure Fabric to use single-transaction blocks. All measurements were done at the load of 10 parallel clients.

At the point of 0 threads, pipelined execution is deactivated. Still, StreamChains throughput is 400 Ops/s higher. That can be explained with block caching and the minor improvements that we realize in the validation phase. Enabling the pipeline leads to a rapid rise in throughput to 2600 Ops/s at a single thread in the first pipeline stage and eventually reaches almost 3500 Ops/s after 2 threads. As a consequence, with the help of the practices of efficient stream processing, together with minor improvements, we are able to push throughput up by more than a factor of 2.

For the following measurements, we fix a number of 16 threads in the VSCC validation and move the peer data back onto hard-drive. We further observe the throughput achieved by the batching of disk writes that we implement at the update of ledger and State DB. The measured throughput at an increasing batch size is compared to a baseline throughput of the last experiment, where we backed the peer by memory (Figure 5.2).

The batch size of 1 is equivalent to the behavior in the original Fabric, where every transaction is immediately written out to disk at the end of the validation phase (in case of single-transaction blocks). The throughput of 60 Ops is, thanks to the pipeline and further optimizations, twice as high as the value we measured for Fabric in chapter 3 (28 Ops/s). Since the batched writes amortize the costs of disk writes, throughput is increasing proportionally to the batch-size, until it reaches 50, where it starts to converge

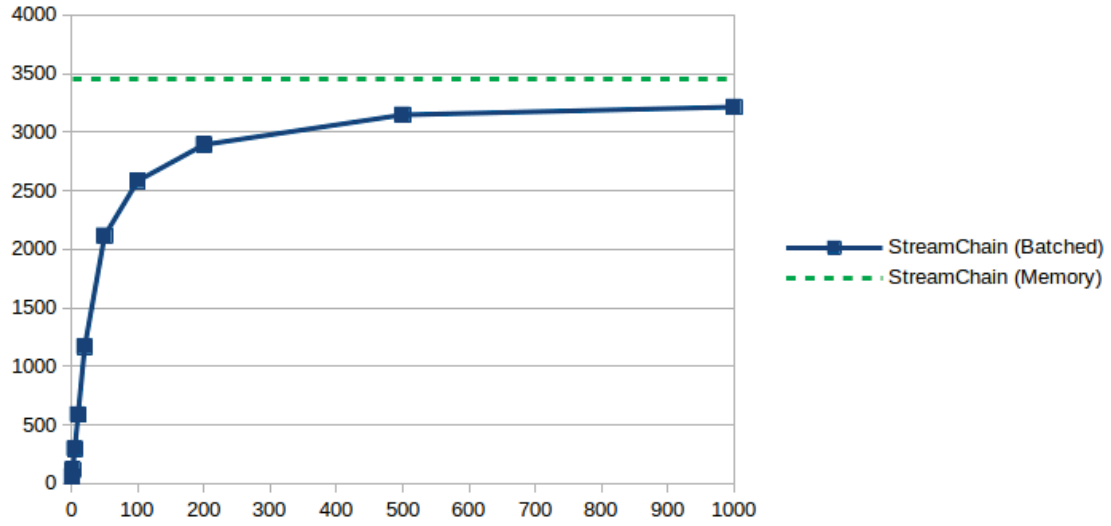

Figure 5.2: Throughput with different Batch Sizes in Commit Step

against an upper bound. At a batch size of 1000, we reach the maximum throughput at around 3200 Ops/s. At this point, we can approximate the throughput of "free disk writes" to memory by an offset of less than 300 Ops/s, which is even less than 10% of the in-memory total. Depending on the application, having an inconsistent ledger state during 1000 operations might not be desirable. However, we already reach 75% of the in-memory baseline at a batch size of 100 and more than 80% for 200 transactions. A suitable tradeoff between consistency and performance might be chosen due to the systems requirements.

The pipelined execution of the validation phase, together with the internal usage of a thread pool proofs to be efficient and increases throughput significantly. The selective batching of transactions, not in the original form of blocks, but inside the commit logic of the peer, manages to amortize expensive writes effectively, and quickly reaches reasonable approximations of the baseline for free disk writes.

5.2 Latency

Having analyzed the throughput characteristics of our implementation, we now evaluate its latency. For that purpose, we look at two graphs showing the relationship between throughput and latency, first comparing StreamChain with Fabric, and secondly separately for every of the phases of StreamChain. Then, we analyze a breakdown of the latencies and how it differs from Fabric.

In Figure 5.3 we show the end-to-end latencies for the three different series on a logarithmic scale. The first is our StreamChain implementation which runs on disk with the batch size of 500 at the peer. The second series is Fabric, completely running

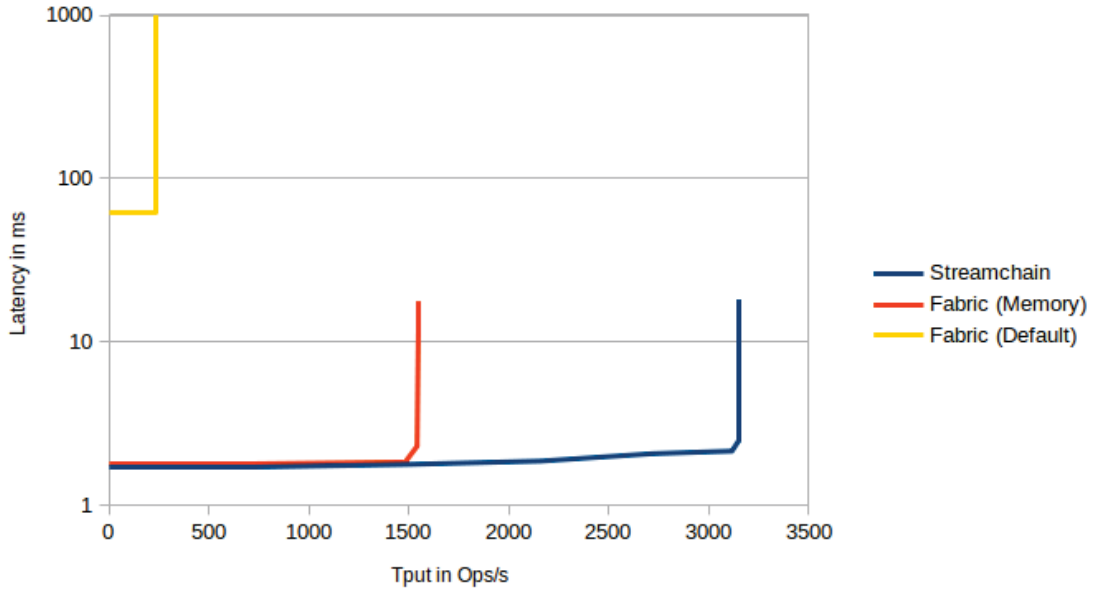

Figure 5.3: Latency/Throughput Comparison StreamChain and Fabric

in memory with single-transaction blocks as well. As a reference, we also include Fabric with the default configuration of block-size 10 on disk as a third series. The shown latencies are determined as the aggregates of average latencies from endorsement, ordering and validation.

With Fabric running on disk, throughput quickly reaches its maximum at 230 Ops/s, bound by the disk writing time of a block. Under low load, latency starts at a time of more than 60 ms, which is, as seen in chapter 3, mainly dominated by the ordering phase. Fabric with a transaction size of 1 and without disk writing overhead starts at 1,8 ms and increases slightly to 2.5 ms at a throughput of 1550 Ops/s, where it arrives at its maximum. The latency of StreamChain is close and even marginally lower. At low load we measure 1,6 ms. Afterwards, it increases to the same maximum value of 2,5 ms as in the case of Fabric in memory, just before it reaches a maximum throughput of 3200 Ops/s. The comparison of the three graphs shows that StreamChain is able to maintain a low end-to-end latency and at the same time offers a throughput that is significantly higher than that of Fabric in its default configuration. The latencies are equal and even lower than in a configuration of Fabric with free disk writes and the smallest block size, which is thanks to the implemented improvements, like block caching.

Next, we go inside StreamChain and illustrate how latency and throughput relate in each of the three phases of endorsement, ordering and validation. The corresponding diagram can be seen in Figure 5.4. We also include a graph for the total latency and throughput, as the one we saw in the previous results.

For low load on the system, the latencies of the three phases are in the same order of 0,5-0,6 ms. The validation phase as the bottleneck of the system, shows the lowest

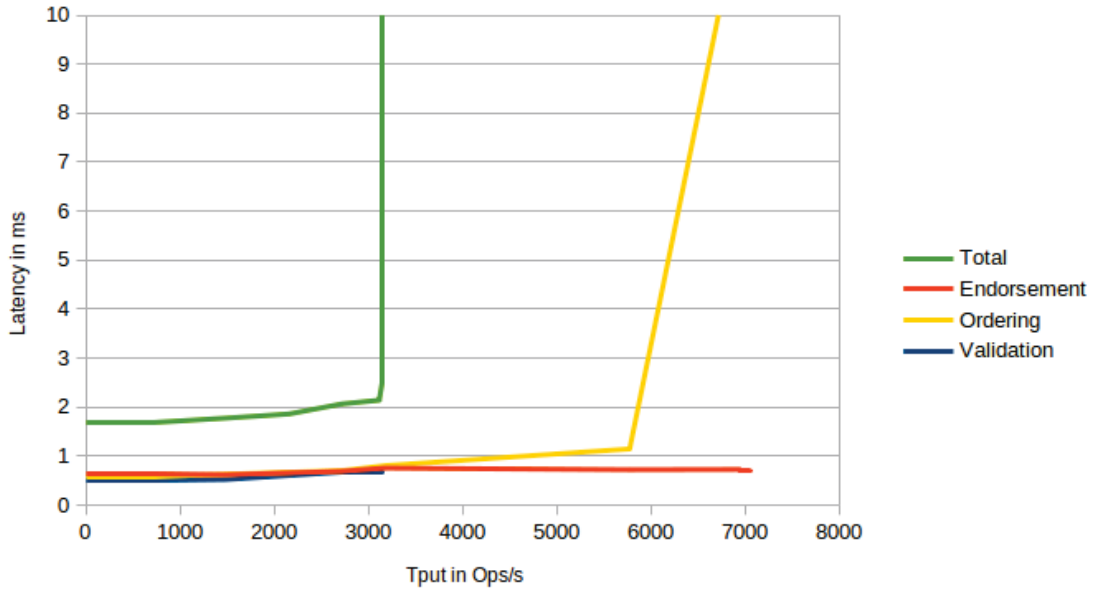

Figure 5.4: Latency/Throughput StreamChain

latency and takes 0,7 ms at maximum throughput. This low increase indicates that the high parallelization inside that phase handles even high loads efficiently. Ordering and Endorsement are measured in a closed loop, so that their throughput limits are the same. Looking at ordering latency, it can be seen that a rapid increase starts at around 6000 Ops/s, while latency of endorsement barely increases. The rising total latency at the point of maximum throughput is therefore caused by the ordering service, which can't keep up with the high load and starts to queue incoming ordering requests. Also, we already know from the measurements in chapter 3, that the endorsement throughput exceeds the one of ordering significantly. After the validation phase, we consequently expect the ordering phase with a maximum throughput of 7000 Ops/s to be the bottleneck of the system.

In the following, we take a look at the distribution of the time spent inside the system. We compare StreamChain with the original Fabric at block sizes 1 and 10, each without disk writes, to see how our changes effect the distribution of latency compared to the original implementation. The results can be seen in Figure 5.5.

The changes introduced in our implementation only have minor impact on endorsement and ordering thus the latencies are the same when compared to Fabric and Tx1. The ordering time is more then 10-times higher for Tx10, and, as we saw before in chapter 3, claims around 80% of the total time. StreamChain shows the biggest difference in the second and third step of the validation phase. The reduced time of RW check can be explained with the minor improvement of the removed read-in of a configuration parameter. The time of the commit step is slightly higher in StreamChain, because disk-writes are amortized, while writes to memory can only be approximated. As expected, the times for the steps of VSCC, RW check and commit are each higher for

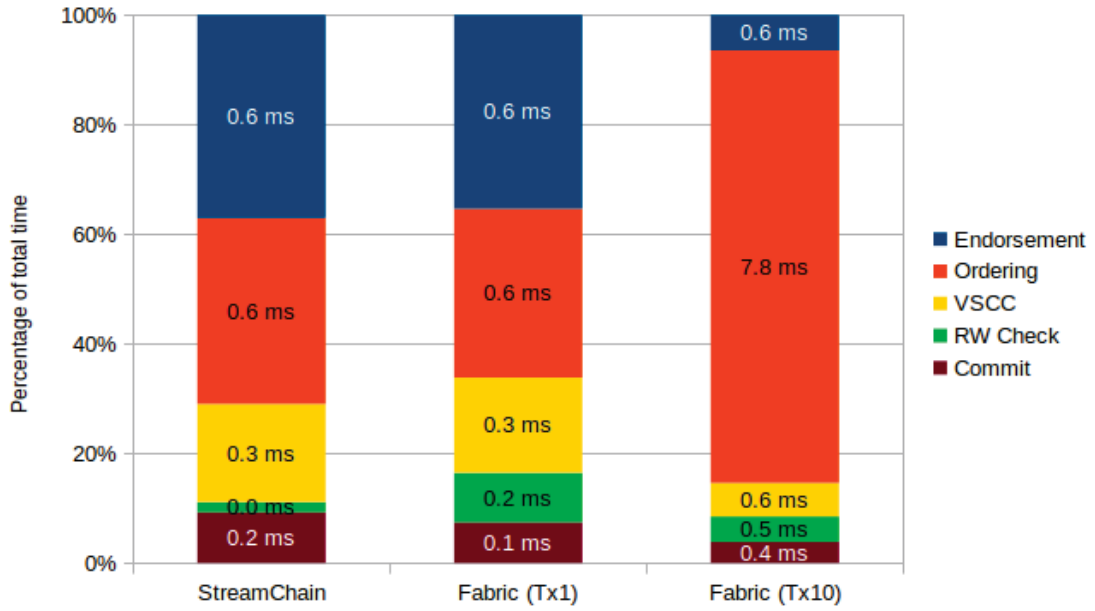

Figure 5.5: Latency Breakdown of StreamChain and Fabric

blocks of 10 transactions. While the distribution of latency is very equal between the three phases in the case of Tx1, the parts of the time that is spent for endorsement and ordering are insignificantly higher in StreamChain.

Processing single transactions instead of batching them into blocks leads to a highly reduced latency, if all data is written to memory. By internal batching and other internal improvements, StreamChain manages to reduce latency even further while committing data to disk.

5.3 Goodput

In the second stage of the validation phase, the read-write sets of each block's transactions are inspected for serializability. Transactions which operate on "dirty" data entries, which were updated in the meantime, are declared as invalid. In this section, we look at the goodput, being the number of operations which end up as valid transactions in the ledger, per second. We compare StreamChain's goodput with the one of Fabric at single blocks and in-memory.

For this experiment, we modify the workload, so that it now contains a 50% ratio of reads to updates. Additionally we run three different variations of this workload, where the size of the key-space, i.e., the subset of keys in the workload which are read and updated, is varied between 10.000 (the whole key-space), 1000 (10% of the key-space) and 100 (1% of the key-space). Since the probability that a specific key is touched by a read or update operation increases, narrowing down the key-space directly results in

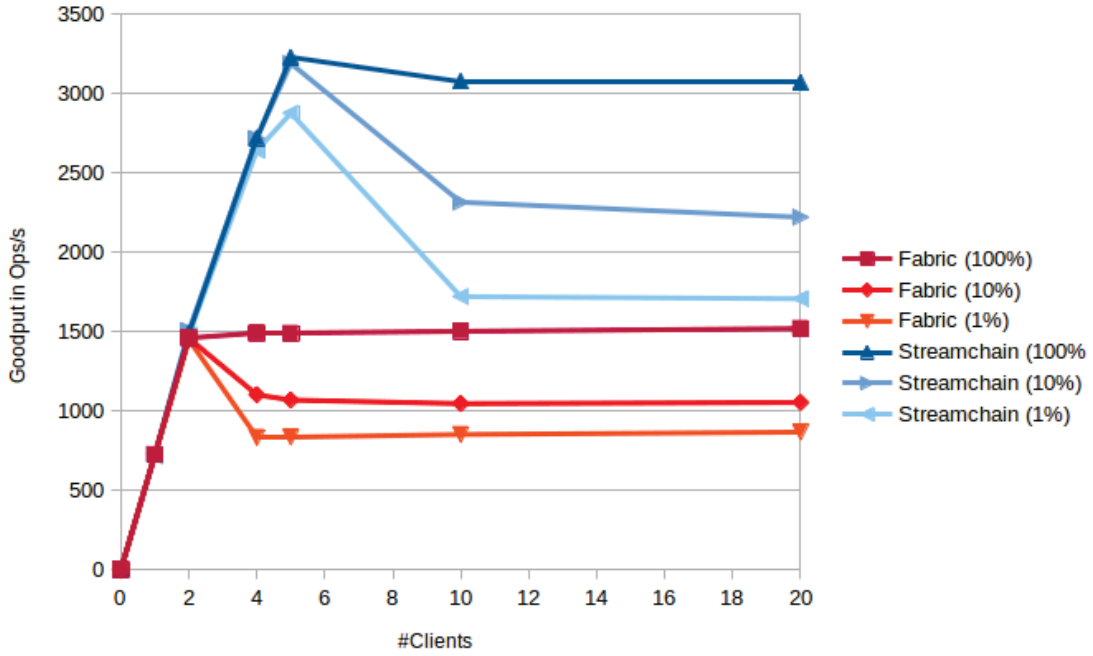

Figure 5.6: Goodput in StreamChain and Fabric

higher rates of dirty reads and consequently, invalid transactions. In 5.6 we show the three series for each StreamChain and Fabric.

In all of the cases, StreamChain can keep a higher goodput than Fabric. Even for the smallest key-space, it falls to 1700 Ops/s, which is still 200 Ops/s higher than Fabric operating on the whole key-space.

5.4 Conclusions

In this chapter, we performed a series of experiments with the goal of evaluating the performance of our implementation of StreamChain and comparing it to Fabric in its original form.

With the help of the pipelined execution and other modifications that favor the throughput of the validation phase, we are able to reach a throughput that is more than twice as high as Fabric with the same configuration of single blocks and in-memory storage. The internal batching of disk writes turns out to be effective, since it approximates the best-case-throughput of free disk-writes by more than 90%. As expected, processing transactions in a stream leads to a significantly decreased latency compared to the default behavior of batching to blocks. As a matter of fact, the latency of StreamChain shows equal and mostly even less end-to-end latency than in-memory Fabric. Since the majority of time is now spent in each endorsement and ordering phase, the

focus of further latency improvements potentially moves towards these two. Next to a high throughput, StreamChain further proofs to maintain a goodput which beats Fabric, even if the set of keys that is operated on is drastically decreased.

Chapter 6

Conclusions

In this work, we addressed the problem of high latencies in permissioned distributed ledgers. Based on Hyperledger Fabric, we implemented StreamChain, where we process transactions in a stream instead of blocks. The latter act as a measure to amortize expensive disk writes, but they stale the transaction and introduce latency into the system. However, minimizing them in size leads to a dropping throughput. We countered this issue by migrating batching mechanisms at block boundaries to lower levels of the systems and execute the validation phase in a pipelined fashion. Beyond that, we optimize our implementation by the means of caching and small improvements in Fabric's validation code.

With our changes in place, we arrive at a solution that offers both low latencies and acceptable throughput. In Figure 6.1 we see a an assignment of StreamChain, together with different configurations of Fabric, to their lowest end-to-end latency at highest throughput. The results show that the latency of our solution is almost two orders of magnitude lower than in Fabric running on disk. We even achieve comparable or lower latencies and twice as high throughputs as when we remove disk-writes as the biggest source of latency from Fabric.

After extensively optimizing the aspect of transaction validation, the time for execution and ordering of transactions turn out to dominate the time in the system and may come to the fore in efforts to push latency down to the sub-millisecond level. The establishment of consensus inside the orderer can be subject to various techniques that we already encounter in classical distributed databases. One possibility is the usage of hardware acceleration, e.g. in the form of an FPGA cluster, to reduce ordering latency close to the baseline of a solo orderer, i.e., without consensus overhead.

The achieved latencies in the order of 1-2 ms open the door for a wide range of new applications of permissioned blockchains. Their latencies may be feasible for the application as tamper-free and distributed alternative to classical centralized databases. A possible use-case can be found in areas like supply chain management, where we usually find a big number of participating parties that exchange data with one another.

Figure 6.1: StreamChain beats Fabric in both Latency and Throughput

Bibliography

[1] G. Wood, "A secure decentralized generalized distributed ledger", *Ethereum*, 2018, ISSN: 1098-6596. DOI: 10.1017/CBO9781107415324.004. arXiv: arXiv:1011.1669v3. [Online]. Available: https://pdfs.semanticscholar.org/ee5f/d86e5210b2b59f932a131fda164f030f915e.pdf?_ga=2.100695233.1575213835.1554832014-965703976.1552935575.

[2] NEO Team, *NEO Smart Economy*. [Online]. Available: https://neo.org (visited on 06/26/2019).

[3] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, S. Muralidharan, C. Murthy, B. Nguyen, M. Sethi, G. Singh, K. Smith, A. Sorniotti, C. Stathakopoulou, M. Vukolić, S. W. Cocco, and J. Yellick, "Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains", 2018. DOI: 10.1145/3190508.3190538. arXiv: 1801.10228. [Online]. Available: http://arxiv.org/abs/1801.10228%0Ahttp://dx.doi.org/10.1145/3190508.3190538.

[4] *Quorum*. [Online]. Available: https://www.goquorum.com (visited on 06/26/2019).

[5] P.-L. Aublin, R. Guerraoui, N. Knežević, V. Quéma, and M. Vukolić, "The Next 700 BFT Protocols", *ACM Transactions on Computer Systems*, vol. 32, no. 4, pp. 1–45, 2015, ISSN: 07342071. DOI: 10.1145/2658994.

[6] S. Nakamoto, "Bitcoin: A Peer-to-Peer Electronic Cash System", *Cryptography Mailing list at https://metzdowd.com*, www.bitcoin.org, 2009. [Online]. Available: www.bitcoin.org.

[7] X. Xu, I. Weber, M. Staples, L. Zhu, J. Bosch, L. Bass, C. Pautasso, and P. Rimba, "A Taxonomy of Blockchain-Based Systems for Architecture Design", *Proceedings - 2017 IEEE International Conference on Software Architecture, ICSA 2017*, pp. 243–252, 2017. DOI: 10.1109/ICSA.2017.33.

[8] IBM, *Hyperledger Fabric: About*. [Online]. Available: https://www.hyperledger.org/about (visited on 06/24/2019).

[9] ——, *Hyperledger Fabric Documentation*. [Online]. Available: https://hyperledger-fabric.readthedocs.io (visited on 06/24/2019).

[10] A. Sharma, F. M. Schuhknecht, D. Agrawal, and J. Dittrich, "How to Databasify a Blockchain: the Case of Hyperledger Fabric", 2018. arXiv: 1810.13177. [Online]. Available: http://arxiv.org/abs/1810.13177.

[11] J. Kreps, N. Narkhede, J. Rao, *et al.*, "Kafka: A distributed messaging system for log processing", in *Proceedings of the NetDB*, 2011, pp. 1–7.

[12] T. T. A. Dinh, J. Wang, G. Chen, R. Liu, B. C. Ooi, and K.-L. Tan, "BLOCKBENCH: A Framework for Analyzing Private Blockchains", 2017. arXiv: 1703.04057. [Online]. Available: http://arxiv.org/abs/1703.04057.

[13] Z. István, A. Sorniotti, and M. Vukolić, "StreamChain: Do Blockchains Need Blocks?", 2018. arXiv: 1808.08406. [Online]. Available: http://arxiv.org/abs/1808.08406.

[14] C. Gorenflo, S. Lee, L. Golab, and S. Keshav, "FastFabric: Scaling Hyperledger Fabric to 20,000 Transactions per Second", 2019. arXiv: 1901.00910. [Online]. Available: http://arxiv.org/abs/1901.00910.

[15] Google, *gRPC*. [Online]. Available: https://grpc.io (visited on 06/24/2019).

[16] ——, *Protocol Buffers*. [Online]. Available: https://developers.google.com/protocol-buffers/ (visited on 06/24/2019).