Department of Informatics
Technical University of Munich

TECHNICAL UNIVERSITY OF MUNICH

DEPARTMENT OF INFORMATICS

MASTER'S THESIS IN INFORMATICS

Experimental Performance Evaluation of Private Distributed Ledger Implementations

Hendrik Folke Leppelsack

Technical University of Munich

Department of Informatics

Master's Thesis in Informatics

Experimental Performance Evaluation of Private Distributed Ledger Implementations

Experimentelle Performanz Evaluierung von privaten Distributed Ledger Implementierungen

Author:	Hendrik Folke Leppelsack
Supervisor:	Prof. Dr.-Ing. Georg Carle
Advisor:	Dr. Holger Kinkelin
	Stefan Liebald M.Sc.
	Dr. Fabien Geyer
Date:	August 9, 2018

I confirm that this Master's Thesis is my own work and I have documented all sources and material used.

Garching, August 9, 2018

Location, Date

Signature

Abstract

Distributed ledger technology (DLT) maintains a shared ledger over a peer-to-peer network with the help of consensus mechanisms and smart contracts. Due to this technology's unique features, it provides the foundation for new trust models and business opportunities. For this reason, DLT is an emerging technology with growing application areas in fintech, government, and healthcare. However, due to the complex interplay of peers, the overall performance of DLT is more difficult to assess compared to a centralized system.

Performance can be measured by different metrics and it can be influenced by multiple factors. To facilitate further considerations, the complex DLT architecture is first divided into four layers: the network, node, ledger, and application layer. For each layer, multiple metrics and influencing factors are defined. Lastly, the concept of elementary and simulative DLT workloads is introduced.

Based on this analysis, a framework is designed which sets up a distributed ledger in a test environment and performs reproducible measurements. This framework is designed such that both, evaluated technology and test environment, are easily exchangeable. The framework design is then implemented utilizing the DLT benchmarking tool Caliper.

As an exemplary matter for performance measurements and evaluation, Hyperledger Fabric is selected, a DLT implementation by the Linux Foundation. Experiments were performed in a controlled lab environment. The evaluation of the measurement results gives insights into the performance impact of four factors, namely variations of transaction rate, workload, block size, and also the effect of packet loss. The measurements show, that factors on each layer can directly impair the performance of the complete network, through increased transaction rate, demanding workloads, unfavorable ledger configuration, or adverse network circumstances.

Zusammenfassung

Distributed-Ledger-Technologie (DLT) führt einen gemeinsamen Ledger über ein Peer-to-Peer-Netzwerk mit Hilfe von Konsensmechanismen und intelligenten Verträgen. Aufgrund der einzigartigen Eigenschaften dieser Technologie bietet sie die Grundlage für neue Vertrauensmodelle und Geschäftsmöglichkeiten. Aus diesem Grund ist DLT eine aufstrebende Technologie mit wachsenden Anwendungsbereichen in den Bereichen Fintech, Gesundheitswesen oder auch für staatliche Organisationen. Allerdings ist aufgrund des komplexen Zusammenspiels der Peers die Performanz von DLT schwieriger zu beurteilen als bei zentralisierten Systemen.

Die Leistung kann durch verschiedene Metriken gemessen und durch unterschiedliche Faktoren beeinflusst werden. Um weitere Überlegungen zu erleichtern, wird die komplexe DLT-Architektur zunächst in vier Schichten unterteilt: Netzwerk-, Knoten-, Ledger- und Anwendungsschicht. Für jede Schicht werden mehrere Metriken und Einflussfaktoren definiert und das Konzept der elementaren und simulativen DLT-Arbeitslasten wird eingeführt.

Basierend auf dieser Analyse wird ein Framework entworfen, das einen verteilten Ledger in einer Testumgebung aufbaut und reproduzierbare Messungen durchführt. Dieses Framework ist so konzipiert, dass sowohl die evaluierte Technologie als auch die Testumgebung leicht austauschbar sind. Das Framework-Design wird infolgedessen mit dem DLT-Benchmarking-Tool Caliper umgesetzt.

Als Beispiel zur Leistungsmessung und -bewertung wird Hyperledger Fabric ausgewählt, eine DLT-Implementierung der Linux Foundation. Die Experimente wurden in einer kontrollierten Laborumgebung durchgeführt. Die Auswertung der Messergebnisse gibt Aufschluss über die Performance-Auswirkungen von vier Faktoren, insbesondere Veränderung der Transaktionsrate, der Arbeitslast, der Blockgröße und auch der Auswirkung von Paketverlusten. Die Messungen zeigen, dass Faktoren jeder Schicht die Performance des gesamten Netzwerks direkt beeinträchtigen können, durch erhöhte Transaktionsrate, anspruchsvolle Arbeitslasten, ungünstige Ledger-Konfiguration oder unvorteilhafte Netzwerkbedingungen.

Acknowledgments

I would like to thank my thesis advisors Dr. Holger Kinkelin, Stefan Liebald M.Sc., and Dr. Fabien Geyer for the continuous support. Their advice and expert knowledge helped to steer this work into the right direction whenever needed.

My sincere gratitude also goes to Prof. Dr.-Ing. Georg Carle for supervising my work and providing me with the opportunity to experience the scientific work at his chair.

Special gratitude goes to my family for their continuous support throughout this work and over the course of my studies.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

Blockchains promise to revolutionize governments [1], fintech [2], and many other fields. This novel technology provides characteristics like immutability and decentralization through consensus mechanisms, which offer a multitude of use cases. Currently there mostly only exist prototypes. Large technologies like Ethereum are missing large-scale real-world applications. While there is a lot of work put into advancing this technology, there are still many technical challenges in different areas like privacy [3], scalability [4] or performance. Advances in these areas are also required for enterprise applications to become feasible. In enterprise environments and especially fintech mostly permissioned distributed ledger technologies are utilized. Instead of allowing access to anyone, like Bitcoin or Ethereum do, these technologies restrict access to a single organization or consortium. Distributed ledger technologies gained traction in 2015 with the founding of multiple projects in this area such as R3[1] and Hyperledger[2]. Often their use has to be justified over centralized technologies. A major disadvantage of distributed applications like DLT is their performance compared to centralized approaches. To make distributed ledger technology competitive it is necessary to optimize this aspect.

This thesis focuses on the evaluation of performance of DLTs in an environment that represents a typical usage. Especially the influence of different network characteristics has not been studied yet, and this thesis tries to give insights in this area. Specifically this thesis aims to solve the following research questions.

[1] https://www.r3.com/

[2] https://www.hyperledger.org/

1.1 Research Questions

Which metrics are relevant to measure performance of distributed ledgers?
Section 3.2 goes into detail about which variables should be measured in a testbed to allow insights into the performance of distributed ledgers.

What are the bottlenecks of distributed ledgers?
Which characteristics and which layer of the network creates a bottleneck to the performance in the network? Therefore in chapter 7 a few measurements are executed which allow an assessment of this question.

What are the limits of performance evaluation inside a testbed?
Section 8.2 interprets the results of the measurements and tries to answer this question.

Is it possible to predict performance based on network and ledger parameters?
Insights into how parameters on different layers influence the overall performance can help to decide if DLT is suitable for specific applications. This question is also investigated in section 8.2.

1.2 Methodology & Structure

In the beginning of this thesis, it has been assessed what constitutes performance in a DLT environment and how to measure it. This as well as the question, which factors might influence the performance, are analyzed in section 3.2. During investigating this, related work has been studied, which is listed and discussed in section 4. Based on the gathered information a design of a framework was developed, which is elaborated on in chapter 5. It was designed to allow deployment of a DLT with different parameters into a configurable environment and taking into account different metrics. The framework is easily extensible and variable to allow exchanging the DLT, for example.

Alongside the theoretical part of this thesis, a lot of work was put into familiarizing and experimenting with the environment and applications utilized in this thesis. Section 2.2 gives insights into the chosen ledger technology Hyperledger Fabric. First efforts were made with Hyperledger Fabric to understand the technology and its deployment process. While the progress stagnated at this point due to Hyperledger Fabric being a young software project and thus quickly evolving and changing, it was possible to deploy to the iLab testbed after a while. The implementation details of this procedure are described in detail in section 6.2. The chapter 6 overall describes the complete realization of the framework, including the just mentioned deployment as well as the other phases decided upon in chapter 5.

With the help of the framework's implementation, a few exemplary experiments have been executed in a test environment. These measurements are detailed in chapter 7, which evaluates four different experiments and interprets their results.

This thesis is concluded by chapter 8 which gives a final overview of the work done and its outcome. Finally it provides an outlook onto future work.

Chapter 2

Background

To get a general understanding of the utilized technologies and tools, the following gives an introduction into DLT and especially Hyperledger Fabric.

2.1 Distributed Ledger Technology

Ledgers have been used for centuries to record transactions for accounting. Digital ledgers are used for bookkeeping and play an important role in the financial sector. One desired characteristic of ledgers is immutability to prevent any falsification. This matches perfectly with the attributes of blockchain. Thus, today's digital ledgers are often implemented utilizing concepts taken over from blockchains.

With the help of blockchains it is possible to extend the ledger concept to a distributed system. Such a system gets rid of the central entity maintaining the ledger. Instead it is possible to have multiple entities / parties / stakeholders watch over the ledger.

Decentralization opens the possibility for a different trust model. While a centralized ledger requires every participant to trust a single entity, decentralization removes this constraint. Through consensus mechanisms it is possible that multiple entities decide on the progression of the ledger. One known way to enable consensus is the Proof-of-Work approach of Bitcoin. Mining with the help of Proof-of-Work is usually used in open systems like Bitcoin, where the trust is minimal between the participants. Unpermissioned ledgers allow anyone to participate, while permissioned ledgers restrict access. Permissioned ledgers only allow specified parties to append transactions and to access the ledger at all. Because of this restriction the participants often know each other, which does not require a trust model as strict as with most unpermissioned ledgers. Specific attacks like spamming the ledger are not as relevant, and thus softened or even completely different consensus mechanisms are possible in this environment.

The consensus achieves, that transactions are added to the ledger in a way every participant agrees on in the end. The goal is that every participant has an identical copy of the ledger with the same transactions, containing the same details, and in the same order.

In addition to transactions, distributed ledger technologies often also support a concept called smart contracts. Cryptocurrencies like Bitcoin support the transfer of assets between entities in the network, while other technologies go further. With the help of smart contracts it is possible to not just perform basic transactions, but even more complex operations. The most popular unpermissioned blockchain with smart contract support is Ethereum which allows to define contracts through code instead of a legal framework. For example smart contracts provide the option to define a transaction that is only executed after some condition has been met. An exemplary application for this would be to bind a transaction of cryptocurrency to the transfer of a physical product. Other systems like a centralized database system have the disadvantage, that this logic and its validation would have to be defined on the application layer. Ledgers with smart contracts allow to define rules within them, natively supporting representation of legal contracts through code.

2.2 Hyperledger Fabric

Hyperledger Fabric is a permissioned distributed ledger framework implementation. [5] describes it as a distributed operating system because of its extensible and modular architecture. Fabric is built to support exchanging specific components of the technology like authentication mechanism or consensus protocol. This allows modification of the framework to fit different use cases. Fabric as well as the whole Hyperledger project is run by the Linux Foundation and has a large open source community, which allows the young project to grow quickly. This has also led to multiple enterprise applications, like a DLT prototype for reinsurance contracts by "B3i"[1].

[1] https://b3i.tech

```go
type SimpleChaincode struct{}

func (t *SimpleChaincode) Init(stub shim.ChaincodeStubInterface)
 ↪  pb.Response {
        return shim.Success(nil)
}

func (t *SimpleChaincode) Invoke(stub shim.ChaincodeStubInterface)
 ↪  pb.Response {
        function, args := stub.GetFunctionAndParameters()
        if function == "get" {
                result, _ := stub.GetState(args[0])
                return shim.Success(result)
        }
        if function == "set" {
                stub.PutState(args[0], []byte(args[1]))
                return shim.Success(nil)
        }

        return shim.Error("Invalid function")
}
```

Listing 1: Sample chaincode excerpt

To give an overview of Fabric, there are a few concepts that shall be introduced. While most of the following is valid for all Hyperledger Fabric versions, this thesis specifically works with version 1.1.

2.2.1 Chaincodes

A basic building block of most DLT are smart contracts, which are called chaincodes in Fabric. They are executed in the network to modify the ledger's state. Golang and Node.js are currently supported for chaincode languages. Listing 1 gives an exemplary chaincode, which implements operations to get and set values in the ledger's state. Chaincodes have to support two public functions. The `Init()` function is called when the chaincode is deployed, while the `Invoke()` function is executed every time the chaincode is called. Chaincode invocation accepts a function name, like in this case either "get" or "set", as well as a list of arguments. These arguments are utilized in this example to get or set the value of any key.

2.2.2 Roles

Hyperledger Fabric is not a homogeneous peer to peer system where each participating node has the same role. Instead different functions of the ledger are distributed among the nodes. The role most nodes in a Fabric network belong to is called **peer**. Peers are the main component of the network and each operates a copy of the ledger. This copy consists of the current state of the ledger as well as its history with the help of a copy of the ledger's blockchain. The peer role is separated into two subroles. **Committing peers** are confined to only maintain the ledger and apply changes committed through the blockchain to their copy. These peers can be queried for the current state of the ledger, but do not take part in transaction creation. **Endorsing peers** have all the capabilities of committing peers, but are also capable of executing chaincode. They accept requests to initiate chaincode executions, which create transactions that may be added to the ledger.

While transactions are created with the help of endorsing peers, they are added to the ledger by the **ordering service**. It gathers, orders, and batches transactions and finally commits them to the blockchain. Defining an explicit order of transactions is required to prevent inconsistencies resulting from race conditions. Batching of transactions is the act of creating blocks instead of committing each transaction individually. While the batching process reduces communication overhead, it generally increases the latency of transactions, which is also observed in the conducted experiments of this thesis.

The ordering service is just the concept, while so-called **orderers** are the actual nodes in the network. Based on the goal of modularity the ordering service is built such that the internal operation can easily be switched out. While Fabric currently supports a single orderer implementation, the issue with this is that it creates a single point of failure. Each transaction has to go through the ordering service and as such failure or attacks can lead to the complete ledger being compromised. Preventing this is possible by distributing the network across multiple nodes. Therefore a protocol is required to reach consensus about which blocks are created. Fabric currently supports an ordering service based on Kafka. Apache Kafka is a distributed streaming platform and is utilized by Fabric to order the transactions in a crash fault tolerant way [6]. While this suffices against node failure, malicious activity cannot be defended against. This is where Fabric currently falls short, but a byzantine fault tolerant implementation of the ordering service is currently in development.

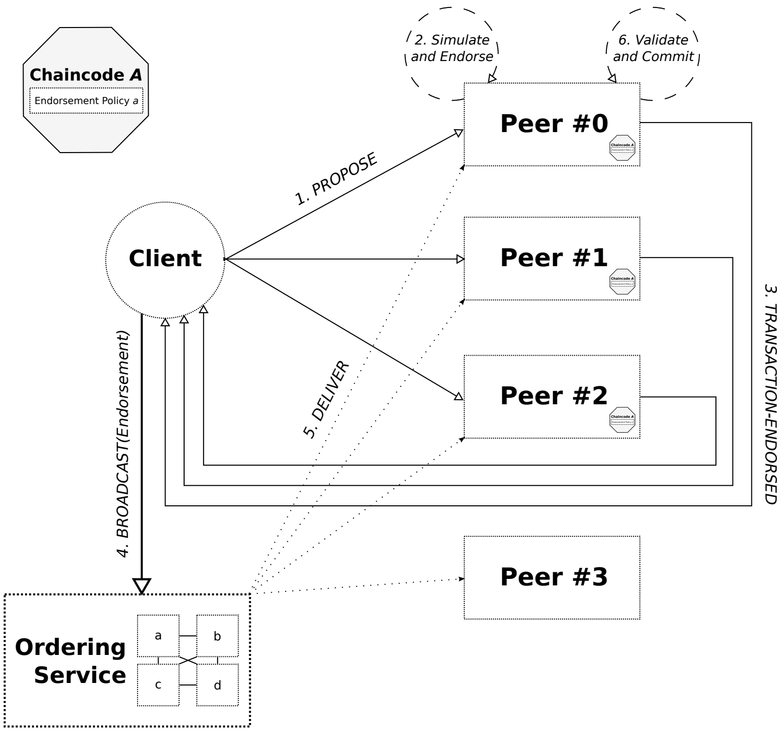

Figure 2.1: Hyperledger Fabric transaction flow [7]

2.2.3 Transaction Flow

In the following it is explained how the described roles interact with each other and how a client is able to cause the addition of a transaction to the ledger.

Usually smart contracts are executed after transactions have been committed to the blockchain. Fabric follows a different approach.

Figure 2.1 shows, that Fabric instead first executes the transaction. Therefore the client creates a transaction proposal which is send to the peers (1). Specifically it only communicates with the peers that are able to simulate its proposal, which requires the corresponding chaincode to be installed on the peer. These so-called endorsing peers simulate the proposal by executing the chaincode on their current state, but not applying the changes (2). The result is the changeset which is signed and returned as an endorsement of the transaction proposal (3). All endorsements are gathered by the client and afterwards forwarded to the ordering service together with the transaction (4). The ordering service creates a definite order of transactions it receives and batches them into blocks. These blocks are delivered to the peers in the network (5), where the transactions are validated and applied to the ledger state (6).

This method is called execute-order-validate and offers determinism. The approach of order-execute, implemented by other DLTs, leads to inconsistent state under specific circumstances. This is solved by the execute-order-validate approach, because after the execution the client can verify itself, that the expected result is returned. Because of the final validation step the client can be sure, that the application of the transaction will either succeed with the predetermined result or fail completely. In neither case the distributed ledger can become inconsistent.

2.2.4 Membership Service Provider

For the enterprise use cases Hyperledger Fabric is utilized for, multiple organizations usually participate in one DLT network. Therefore each organization provides peers in the network. The so called membership service provider (MSP) associates the peers with the organizations, which are represented by cryptographic identities. By default Fabric provides a PKI based implementation of the MSP, where each organization is identified by at least one CA. Members of an organization have certificates that are signed by the organization's CA. In addition to simple membership, there is also a concept of roles, which allows to define administrators of an organization, for example.

2.2.5 Endorsement Policy

These roles, defined by the membership service provider, are important for endorsement policies. They define which roles have to endorse a transaction proposal, such that it is accepted during the validation phase. For example this could be, that at least two admins from different organizations have to endorse a transaction. Otherwise the changeset is not applied, because the final validation step fails.

2.2.6 Channels

Finally Hyperledger Fabric supports a concept called channels which enables confidentiality. A channel is a separate ledger instance, maintaining its own blockchain. Only a subset of peers participates in channels, creating a subnet with its own chaincodes. Invocations of chaincode just influence the state within the channel. Thus channels allow to isolate transactions on different ledgers, resulting in confidentiality and improvement of scalability and performance.

Chapter 3

Analysis

This chapter describes the available options for experiment platforms and analyses possible DLT metrics, factors and workloads.

3.1 Experiment Platforms

There are different kinds of platforms [8] that can be used to evaluate network technologies like DLTs. One of them is **simulation**. Instead of working with real systems and real hardware, this approach simulates interaction of different components, without necessarily imitating the complete network stack. This provides a completely controlled and reproducible environment in which the experiments are conducted. Because there is no direct dependency on hardware and real networks, it is easier to scale the size of the network. The possibility to keep the complete evaluation on a single machine provides easier debugging and even time manipulation, which allows to simulate large loads on the network. These aspects would be more difficult on other evaluation platforms because of their distributed architecture. All these abstractions lead to reduced realism and possibly falsified or biased results. ns-3 [9] and OMNeT++ [10] are examples for network simulation frameworks.

An alternative approach, that gives more realistic results is **emulation**. Emulators like Mininet [11] or Emulab/Netbed [12] provide a hybrid approach with real applications and simulated network components. Thus they increase realism and accuracy of the results, while accepting additional overhead and reducing reproducibility.

The final approach is the utilization of a **testbed**. Utilizing dedicated infrastructure to execute the experiments allows to gather realistic results. The downside is that hardware for a testbed is costly and limits the scalability of the experiments. Also testbeds are scenario specific. Some testbeds are designed to execute wireless experiments, while others are built for internet-scale experiments.

These limitations lead to testbeds often being set up cooperatively, like Grid'5000 [13] which is funded by multiple French universities and institutions and consists of 1000 nodes in a network spanning across France. Sharing the testbed comes with usage restrictions and thus availability limitations for each individual user. Another example is PlanetLab [14] which is a global testbed. Such large networks come with additional challenges, like resource management [15] or security concerns [16].

The platform utilized for this thesis is the Baltikum testbed, extended to support the iLab isles. Because this thesis aims to measure network characteristics and provide highly realistic results, the decision to use a testbed was made. The scenario of private DLT evaluation does not require high scalability and by utilizing a local testbed instead of a geographically distributed testbed controllability and availability is increased.

3.2 DLT Metrics, Factors, Workloads

For both metrics and performance factors, this thesis categorizes by layer. Mainly this divides the items into the layers depicted in figure 3.1.

At the bottom is the network layer, consisting of the underlay network, which resembles the actual network, and the overlay network. The overlay network describes the composition and distribution of ledger nodes on top of the underlying infrastructure.

The nodes themselves build a separate layer describing the characteristics of the hosts. They also execute the ledger which represents its own layer.

The ledger layer itself can again be split into different subgroups depending on the technology providing this layer. In the case of Hyperledger Fabric this would be Client, Endorser, Orderer, and Committer.

Finally on top of all other layers is a last layer which describes the application that is run in the ledger environment.

Performance Metrics

On the **network layer** there is mainly one metric, which is *throughput*. Throughput in this case specifically means the amount of raw packets or bits per second at each individual link.

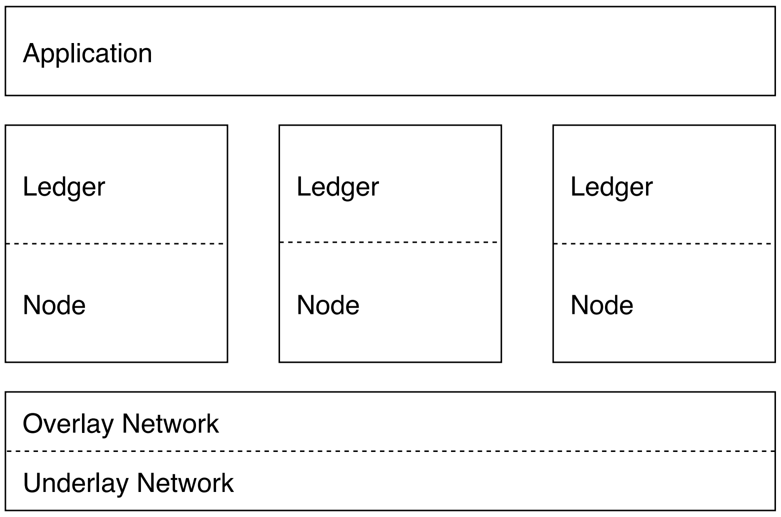

Figure 3.1: Application layers of distributed ledger technologies

On the next level up - on the **node layer** - the *resource usage* of each individual node can be evaluated. This includes CPU, RAM, network, and also persistent storage utilization. Each of these metrics can be measured on the individual nodes and might differ depending on the following layer.

The **ledger layer** is split, as described before, by the roles assigned by the ledger technology. Generally there exist client nodes which provide the main insights into the ledger performance itself. On the client machine it is possible to inspect the *transaction backlog* to observe the amount of pending transactions. Transactions can be finished in two ways, either by the client being notified of the successful addition of the transaction to the ledger, or by failing because of conflicts, network problems, etc. This is taken into account by the metric *success rate*, measuring the overall percentage of successful transactions. Successful transactions can be evaluated on the duration it took to persist in the ledger network, which is represented by the *transaction latency.* Besides full transactions, ledgers also offer a way to read their state. These requests have their own characteristics, for instance measured through *read latency.*

Similarly to the client, some metrics can also be evaluated on other peers in the network. Committing peers have some *transaction throughput*, which describes the amount of transactions that are handled by the nodes per second. In this case it does not matter whether the transaction finally succeeds. This is instead taken into account by the *transaction goodput* metric, which measures the amount of successful transactions per second.

Performance Factors

Beginning bottommost there are a few factors that characterise the **underlaying network**. These include basic properties like *bandwidth* of links, *loss* of packets and *delay* of packets, but also includes characteristics like *churn* which describes the rate at which nodes join or leave the network. In addition the *topology* of the network has a high potential impact on the network by directly or indirectly influencing the previous factors. For example the *diameter* directly impacts the delay through increased hop count.

On top of this the **overlay network** defines a **distribution of roles** across the network. This distribution may create some clustering of nodes of different organizations for example. Different clusters on the overlaying network may be connected through a low bandwidth link on the underlaying network, creating interesting use cases.

The **node layer** affects the performance through *resource constraints*. Nodes in the network can either be homogeneous, with the same constraints on CPU, memory, network, and storage, or might be heterogeneous like in the case of IoT networks where there are low resource devices which interact with a stronger backbone.

Above the node layer the **ledger layer** establishes further factors to the performance. With multiple applications in a network, there might be a varying *distribution of chaincodes* in the network. On each individual peer in the network, different amounts of chaincodes might be installed, which potentially influence the execution of each other. To store the results of the chaincodes there needs to be a state storage implementation. Depending on the utilized technology different *store types* might be supported on the participating peers in the network. This ranges from key-value stores to full-fledged relational databases and also differs by the performance of the actual implementation used. Additionally a main component of each distributed ledger is the *consensus mechanism*, which directly affects which peers have to communicate and how much communication has to take place. The consensus mechanism might be configurable or even exchangeable, which allows to tune the performance of the ledger. Lastly an important property of blockchains, which is in most cases configurable, is the *block size*. It defines how many transactions can be included in each block and thus is another configuration option to tune the ledger.

Finally the **application layer** evidently provides a variety of ways to influence the performance. The *application's task* that shall be solved, has a characteristic impact on the peers and the network. Depending on the task it might be necessary to implement a larger *amount of chaincodes* to solve the problem. In addition a peer might support different *chaincode languages*, where a compiled language performs preferable compared to an interpreted language.

Besides this, applications interact with the ledger in varying expected frequencies. This aspect can be represented by varying the *transactions per second* that are executed in the network. In addition to the speed chaincodes are invoked at, they might be called in a specific order, defining a kind of recurring *transaction flow*. One example use case would be, when a modification of a value in the ledger state requires a verification from a separate party. For this purpose an application would implement two kinds of transactions: one which requests a modification and another which verifies it. In this case it would be expected that the "modification request" and "verification" transactions occur shortly one after another and access a common variable in the ledger.

In addition to these factors there are also technology specific aspects. For example in the case of Hyperledger Fabric there exist endorsement policies which influence the amount of traffic necessary to submit transactions.

To take a closer look at the application level and especially the possible tasks, different kinds of workloads are discussed below.

Workloads

Workloads are a definition of a load that is executed on a ledger network to evaluate its ability to handle a specific use case and its performance therein.

Two kinds of workloads are distinguished, which are based on the definitions in [17]:

Elementary Workloads

Elementary workloads are workloads that are not realistic and would not occur in practice. Instead they aim to put pressure on a single aspect in the network.

The following elementary workloads are mainly basic chaincodes which can be executed in any context and with varying speed:

DoNothing This workload is implemented through the minimal chaincode that still allows transactions to take place. It should not create any overhead on any resources neither on the network nor on individual nodes. Programmatically this workload is equal to a `return true` expression.

CPUHeavy The CPUHeavy workload aims to stress the nodes and especially their processor by executing a computationally intensive task. This workload is preferable to measure the impact of chaincode execution on the performance as it focuses on the nodes instead of the network characteristics like latency.

This workload can be realized through any computationally intensive task while minimizing side effects on other resources, like sorting of an inverted array, CPU-bound Proof of Work algorithms, or a simple empty for-loop as long as the compiler does not run optimizations to remove this code.

DataHeavy The third elementary workload is called DataHeavy. Goal of this workload is to simulate chaincodes, which expect parameters that are large in size. This stresses the network because each execution of the chaincode requires more data to be transmitted to the peers. Besides these parameters the payload is similar to DoNothing, in that it shall not execute any code to prevent other overhead.

Simulative Workloads

Simulative workloads are a basic implementation of a realistic use case. The workload is reduced as far as possible to be a generic implementation of the task it represents. An additional parameterization of the workload allows to figure out the impact of changes in the workload on the overall performance.

Key Value Store
One example for a simulative workload is an implementation of a key value store on the blockchain. Other benchmarks like the YCSB benchmark [18] have run similar workloads on NoSQL databases. A modified version of this benchmark can also be applied to distributed ledgers.

Basically this workload executes a set of operations (read, write, scan, insert) on a previously defined set of records. The amount of records and operations and the proportions of the different kinds of operations can be configured. This allows to generate workloads that emulate realistic scenarios.

Chapter 4

Related Work

In the following, related work is outlined. While papers like [5] make basic measurements, the main focus of the following six papers lies on performance evaluation of DLT.

Blockbench

paper: [17], source code: https://github.com/ooibc88/blockbench

The Blockbench paper describes itself as “the first evaluation framework for analyzing private blockchains”. It is a framework that allows to easily integrate different blockchain technologies and compare them. The paper describes the design of the framework and defines a few metrics and workloads which also inspired some definitions in this thesis. Measured metrics are mainly ledger-based, like transaction latency or throughput, but also extend to measurements on the nodes and network like memory usage and network utilization. The workload definitions are split into macro and micro workloads. The described micro workloads are DoNothing, Analytics, IOHeavy, and CPUHeavy and focus on single aspects and layers of blockchain applications. The macro workloads represent basic but complete applications, like a key-value storage, Smallbank, and a Ponzi scheme. Finally the paper provides some evaluation results from different workloads that have been implemented for the technologies Bitcoin, Hyperledger Fabric, and Parity. It focuses on the comparison between these three platforms and only briefly touches the impact of blockchain-specific configuration options like block size. The deployment and installation architecture and scripts for the technologies are minimalistic but fully suffice for the purpose of the paper.

In summary, this paper gives first detailed insights into the performance of private blockchains and formed the basis for subsequent papers.

Performance Analysis of Private Blockchain Platforms in Varying Workloads

paper: [19]

Hyperledger Fabric and Ethereum are evaluated in this paper as in the paper discussed before. For both technologies it implements an application with different smart contracts. The execution times of the smart contracts are compared. But the main focus lies in varying the amount of transactions that are executed. The paper inspects metrics on the ledger layer, specifically execution time, transaction latency and transaction throughput. These metrics are evaluated for both technologies with varying numbers of transactions.

Overall this allows to compare the performance of different workloads as well as making statements on the pros and cons of the two technologies.

Predicting Latency of Blockchain-Based Systems Using Architectural Modelling and Simulation

paper: [20]

This paper puts a special focus on the transaction latency of blockchains. It does not only measures the performance of technologies, but tries to predict it through modelling. The utilized model is a Palladio Component Model [21] which is specifically designed for performance and reliability modelling. After constructing such a model, the predictions are compared to the results of experimental setup. For evaluation purposes an incident management system is modeled and executed on Ethereum. Ethereum as an exemplary platform has the concept of confirmation blocks. Transactions only count as confirmed if a predefined number of blocks have been appended to the chain since the transaction has been included. This reduces the probability that the transaction's block is part of a sidechain and might be abandoned by the network. Confirmation blocks are a ledger factor that is varied by this paper for both the model and the experimental setup.

In summary, the paper studies the feasibility of modelling blockchain performance and constructs a model that has a median relative error below 10%.

Blockchain Orchestration and Experimentation Framework

paper: [22], source code: https://github.com/wshbair/Blockchain-Orechtration-Framework [sic]

This paper focuses on a specific use case, specifically "Know Your Customer" (KYC). KYC describes the practice of identity verification in the context of banks and their customers. The benefit of introducing blockchain technology to this problem is described as the possibility to share customer data after the first assessment. Additionally it also prevents tampering of the data based on the append-only nature of blockchains. The paper describes a proof of concept for this macro workload implemented in Solidity, which is executed on a private Ethereum network in the testbed.

The testbed is the Grid'5000 platform which is a testbed spanning across France. It is orchestrated with an extended version of the Ruby-Cute[1] tool, which is specifically designed for orchestration of Grid'5000.

Within the testbed the number of mining nodes and the transaction rate are varied to measure its impact on the transaction latency. The results of these measurements allow to tune the blockchain and network parameters.

Overall this publication focuses on the deployment and orchestration in the Grid'5000 platform. While the workload and evaluation is mostly a proof of concept, the paper promises the adoption of additional technologies other than Ethereum in the future.

Performance Benchmarking and Optimizing Hyperledger Fabric Blockchain Platform

paper: [23]

As the title suggests this paper examines performance specifically of Hyperledger Fabric. Restricting themselves allows the authors to consider particular parameters and characteristics of Hyperledger Fabric. Instead of just observing metrics on the network or generic ledger layer, this paper also profiles Fabric and evaluates the internal call stack. Factors that are evaluated also includes exchanging Fabric's state database, which is utilized by the peers to maintain the ledger's state. While this restriction prevents easy extensibility to other DLTs it allows deeper insights into Fabric. This is utilized in this paper to suggest three general optimizations: Caching in the authentication process, parallelization of endorsement validation, and read-write set validation improve the performance significantly. The results of this paper have been integrated into Hyperledger Fabric.

Summarizing, this paper describes the results of benchmarking Hyperledger Fabric and utilizing the outcomes to tune it. Because the paper itself is just tailored to a single technology it does not loose many words on the implementation and actual measurements. It is to be assumed that the implementation is built generically and the authors do not show the intent to extend the work to further technologies. Instead they plan to vary the testing environment through different topologies and experimenting in WAN environments.

Caliper

source code: https://github.com/hyperledger/caliper

Caliper is a recent addition to the Hyperledger projects. It is a benchmark tool especially built for blockchain technology. This project allows to define and execute workloads on a blockchain network. On-chain characteristics and process properties are monitored and aggregated into a report. This report includes metrics like transaction throughput and process resource usage.

[1] https://github.com/ruby-cute/ruby-cute

While this framework currently only supports three Hyperledger blockchain projects, it is built in a modular way to allow easy addition of other technologies. Similarly there are only few workloads implemented at the moment, but these can be easily extended as well. This tool is utilized by this thesis, as described in chapter 6.

Conclusion

Table 4.1 compares the described papers with regard to the aspects relevant to this paper. These studies mostly concentrate on a high level view of performance. They consider ledger layer metrics and most of them take ledger factors into account. Only few [17][22] extend their scope down to the network layer. And while many papers experiment with multiple DLTs, only two [17][22] describe an extensible framework.

The concept of elementary workloads is only employed by [17]. Other papers focus on realistic workloads or even only work with a single one.

The goal of this thesis is to define a framework which is capable of executing arbitrary workloads and is able to influence and measure on all layers, allowing further insights into cause and consequence in the area of DLT performance. The following chapters design, implement, and evaluate the framework, with the exception of node layer factors and metrics and realistic workloads. These aspects have not been fully implemented in the scope of this thesis, but the framework is built to allow easy extension to support them.

	[17]	[19]	[20]	[22]	[23]	Caliper	This Thesis
Framework	✓	✗	✗	✓	✗	✓	✓
Factors							
network	✗	✗	✗	✓	✗	✗	✓
node	✗	✗	✗	✗	✓	✗	~
ledger	✓	✗	✓	✗	✓	✓	✓
application	✓	✓	✗	✓	✓	✓	✓
Metrics							
network	~	✗	✗	✗	✗	~	✓
node	~	✗	✗	✗	✓	✓	~
ledger	✓	✓	✓	✓	✓	✓	✓
Workloads							
elementary	✓	✗	✗	✗	✗	~	✓
realistic	✓	✓	✓	✓	✓	✓	~

Table 4.1: Comparison of related work

Chapter 5

Design

The goal of this chapter is to define a framework which executes experiments on DLTs. It shall support deployment and configuration of the technology as well as the actual measurement and evaluation of the results. This way it is possible to define reproducible experiments with minimal manual overhead. Factors and metrics shall be applicable / measurable on different layers as defined in section 3.2. The designed framework shall also not be restricted to a specific DLT, but shall allow to easily exchange the evaluated technology.

To support the generic approach the following roles are defined:
The **orchestration host** is the host directing the whole experiment. It is not part of the system under test, instead it communicates with the respective hosts during setup and measurement. The orchestration host is where the experiment's parameters are defined and where in the end the results of the measurements are accumulated.

The hosts that actually take part in the experiment are intuitively called **experiment hosts**. These hosts are controlled by the orchestration host and their behaviour and activity is recorded. Two roles exist that are themselves experiment hosts. First are the **ledger hosts**, which actually run the technology that is evaluated with the framework. Secondly there are **benchmark hosts**, which execute the benchmark. These hosts communicate with the ledger hosts to run the workloads on the system. The benchmark host role is separate from the orchestration host to allow measurements on the client-side part of the DLT as well. Additionally this also allows arbitrary placement of the benchmark hosts in the network.

As previously mentioned the orchestration host will not just conduct the experiment but also initiate the setup. To separate the different purposes of the framework, it is split into the three phases, deployment, measurement and evaluation, which will be discussed in the following.

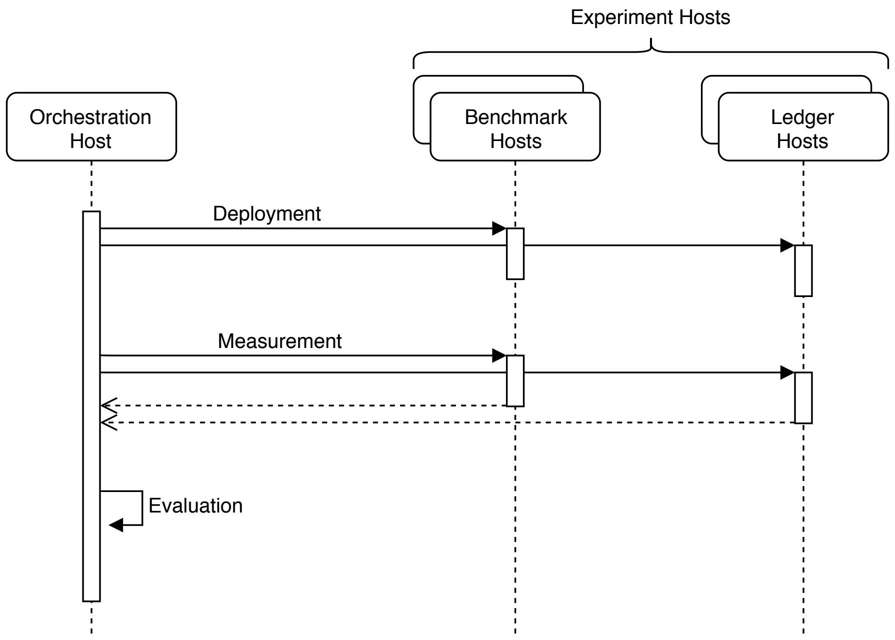

Figure 5.1: Framework Workflow Overview

Figure 5.1 gives an overview of the framework's workflow. The single orchestration host initiates all operations. In case of the deployment and measurement phases, it communicates with all experiment hosts. The first phase installs and preconfigures all dependencies on both the benchmark hosts as well as ledger hosts. During the measurement phase the orchestration host prepares all experiment hosts for the measurement and initiates the measurement. This phase concludes with measurement data being transmitted back from all experiment hosts to the orchestration host. The final evaluation phase happens solely on the orchestration host because it works with all data gathered in the previous step.

The individual phases are described in more detail in the following.

5.1 Deployment Phase

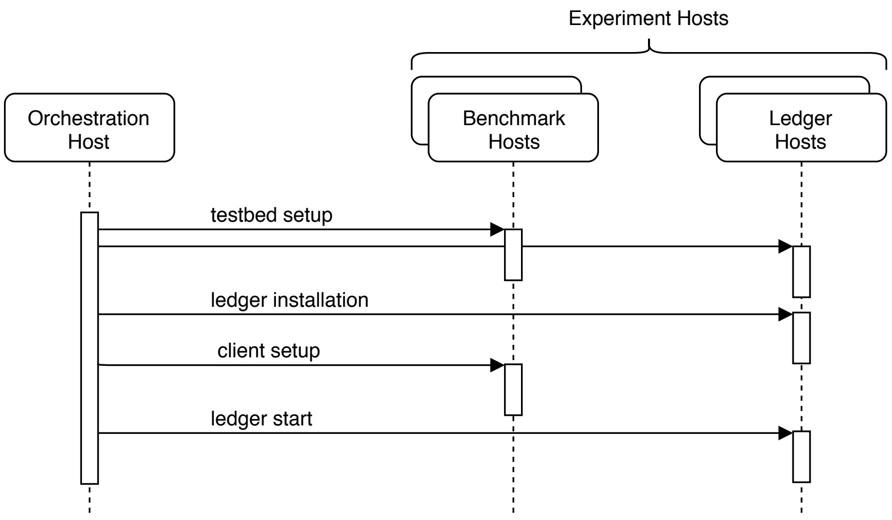

Figure 5.2: Deployment Workflow

Before any measurements take place the testbed has to be prepared. This begins with any testbed specific software installation or configuration to facilitate all following steps. Afterwards, the pristine experiment hosts are equipped with anything that is always required, no matter what DLT is evaluated. This includes tools, which are utilized during the measurement phase to record the hosts' behaviours, for example.

With the general setup steps done, the installation of the ledger technology on the ledger hosts follows. All ledger hosts are supplied with the software they have to run and the configuration file, required to function within the testbed. The same applies to the benchmarking hosts, which the benchmarking application is deployed to. Finally in the deployment phase the DLT is started and the complete system is prepared to undergo measurement. Figure 5.2 represents this workflow graphically.

5.2 Measurement Phase

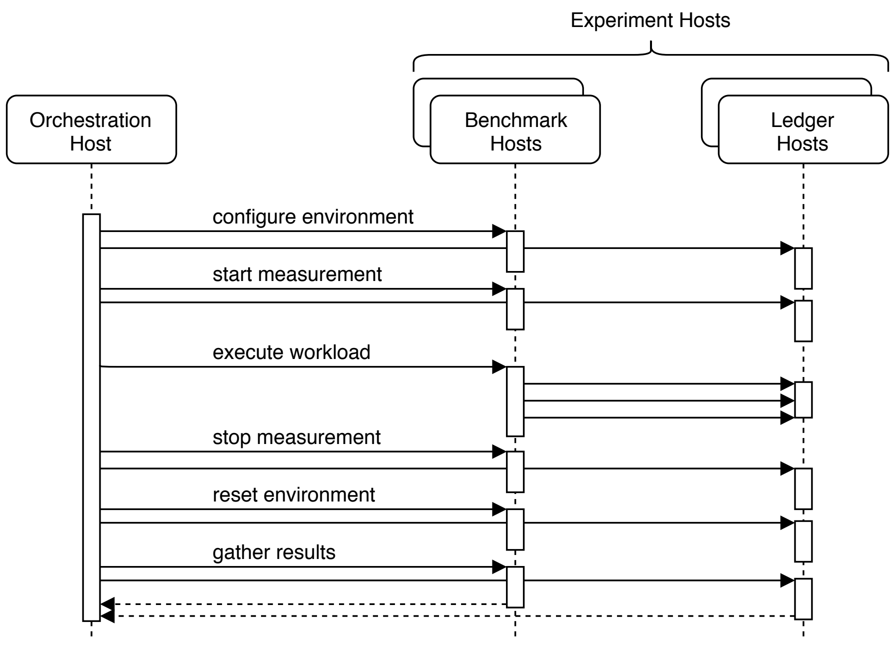

Figure 5.3: Measurement Workflow

Figure 5.3 describes the measurement phase, beginning with an initial configuration step. To run the experiment any final configuration changes are applied to the experiment hosts, like network impairment for example. After this, initialization of all monitors on the experiment hosts takes place. This may include logging of network traffic or resource usage, for example.

Workload execution is then initiated by the orchestration host, which afterwards doesn't interfere with the experiment and waits until its completion. Instead the benchmark hosts take over and operate the experiment based on the workload definition. All previous configuration and initialization steps allow that the system runs the experiment without any external actions. Changes during the experiment are timing based or directly induced by the workload execution on the benchmark hosts. This prevents any distortion of the experiment through management traffic for instance.

Finally after the workload execution finishes, the experiment is shut down. Monitors are stopped on the experiment hosts and any impairments applied during the measurement configuration step is undone. It for example removes artificial network loss which has been put on the network, to allow continued undisturbed work on the testbed.

Lastly any gathered information is extracted from the experiment hosts and gathered on the orchestration host. This allows to immediately reset any hosts in the network but the orchestration host, for subsequent measurements.

5.3 Evaluation Phase

The final phase is the evaluation phase, in which the gathered data has to be evaluated, beginning with a preprocessing phase. The results originating from multiple hosts go through the following steps

- **transformation** to simplify further processing like converting to a common file format
- **cleaning** like deduplication of logged traffic on multiple hosts
- **normalization** by unifying the timesteps for example
- **integration** to connect the different data sources among themselves and to the experiment definitions

This preprocessed data can afterwards be evaluated based on the relevant metrics. A few ways of evaluation are defined in this thesis and the format of preprocessed data allows to easily add further evaluation methods.

Chapter 6

Implementation

This chapter walks through the implementation of the framework designed in chapter 5. In advance the specific testbed, that is utilized in this thesis, is described.

6.1 Experiment Environment

Baltikum Testbed

As described in section 3.1, a testbed experiment platform is chosen for this thesis to allow realistic insights into different layers of DLTs including the network. The Baltikum testbed is a private testbed of the chair of network architectures and services at the Technical University of Munich (TUM). It consists of a few server pairs, available for experimentation purposes. All testnodes are managed from the central server, which is called kaunas. Management of the nodes is done through “pos”, a plain orchestration service especially built for this testbed. It allows to remotely boot the testnodes into system images that are deployed via the network. Additionally it is also capable of deploying and executing scripts on the hosts as well as gathering any experiment results. While this is a great resource to be used for measurements within this thesis, the networks that currently exist with the server pairs do not resemble a typical DLT environment. For realistic compositions two or three nodes are too few, which is why Baltikum has been extended.

iLab

The iLab is a lab course at the network chair of the TUM [24]. It allows students to build and experiment on networks and different protocols, with a combination of e-learning and practical lab exercises. The part of the iLab that is relevant to this thesis is the lab environment. Each student group participating in the course gets access to an “iLab isle”, consisting of six PCs, two switches, and two Cisco routers. The PCs come with the following specifications:

- 4 core CPU (Intel(R) Xeon(R) CPU E3-1265L V2 @ 2.50GHz)
- 16 GB memory
- 1 Gbit/s network
- 120 GB SSD

The lab room contains eight of these isles, of which all but one are accessible to the students. One isle is the so-called remote isle, which is remotely accessible to allow instructors of the course to prepare and test the exercises themselves.

Recently the iLab machines have been added to the Baltikum testbed. This allows experiments consisting of multiple homogeneous nodes, even though these nodes might not be as powerful as the other machines in the Baltikum testbed. While the Baltikum testbed has been regularly used for measurement on the server pairs, this is the first scientific work utilizing the iLab integration.

6.2 Deployment Phase

To work with the Baltikum testbed “pos” is utilized for setup and initialization. This includes booting available nodes into a custom boot image, which is a minimal installation of Debian 9 “stretch”. Afterwards Ansible is used for orchestration. Ansible[1] is an open-source orchestration tool used in DevOps. It allows easy deployment, configuration, and automation and has a big community offering introductory material, support, and a multitude of reusable modules. Ansible has a simple and efficient architecture: It does not rely on any agent software being installed on the controlled hosts. Instead it connects to the machines via SSH and executes short tasks on them. These tasks are called “Ansible Modules” and they range from simple debugging, over file transfer, to execution of custom shell scripts. Ansible defines a large amount of modules and there is the possibility to extend them. Any modules that shall be executed, are defined in “Ansible Playbooks". Playbooks are YAML files with a simple structure, where for each host or group of hosts a list of tasks is defined.

Listing 2 shows an excerpt of an Ansible playbook implementing the testbed-specific initial framework phase: The iLab testbed is by default not set up to connect to the internet. Thus the standard gateway has to be configured for all experiment hosts. Listing 2 defines two tasks, which are part of the `pre_tasks` block, which allows external Ansible roles to be executed afterwards. The first task, beginning in line 3, checks if the standard gateway is already set.

[1] https://www.ansible.com

If this task fails and thus the standard gateway is missing, the next task configures it. The file `/etc/baltikum_ilab_control_ipv4`, which is read in line 8, contains the IP address of the gateway for the baltikum testbed.

```
1  - hosts: experiment-hosts
2    pre_tasks:
3    - name: check for missing standard gateway
4      shell: ip r l | grep 'default via'
5      register: gatewayCheck
6      ignore_errors: True
7    - name: set default gateway
8      shell: ip r a default via $(cat /etc/baltikum_ilab_control_ipv4)
9      when: gatewayCheck is failed
```

Listing 2: Ansible testbed preparation

Another way in which Ansible helps to keep the implementation as generic as possible is through the ability to separately define the hosts that shall be managed. As listing 2 shows playbooks do not require individual hosts by IP address. Instead, in line 1 a group `experiment-hosts` is given that was defined in a so called inventory file. Ansible's concept of an inventory file allows to define hosts independently from the tasks. This way machines can easily be exchanged or added, without rewriting the playbooks. Listing 3 depicts a short inventory file with two hosts which are both part of the group `experiment-hosts`. At the start of the file two aliases are defined for the hosts (lines 1-2), which are later added to the group (lines 4-5).

```
1  pc1    ansible_ssh_host=192.168.0.5
2  pc2    ansible_ssh_host=192.168.0.6
3
4  [experiment-hosts]
5  pc1
6  pc2
```

Listing 3: Sample Ansible inventory

Ansible is run on the orchestration host. In case of this execution this is the same machine that also boots the machines via pos. All other nodes are part of the iLab testbed and resemble the experiment hosts from the design, with the addition of a preparation host role. The preparation host is responsible for the creation of shared data between the experiment hosts, like crypto material.

Using Ansible instead of pos minimizes the dependency on the Baltikum testbed and allows to utilize the range of modules provided by Ansible. In the case of this framework it carries out the deployment and configuration of any ledger and testing software, as well as starting the actual measurement and extracting the results.

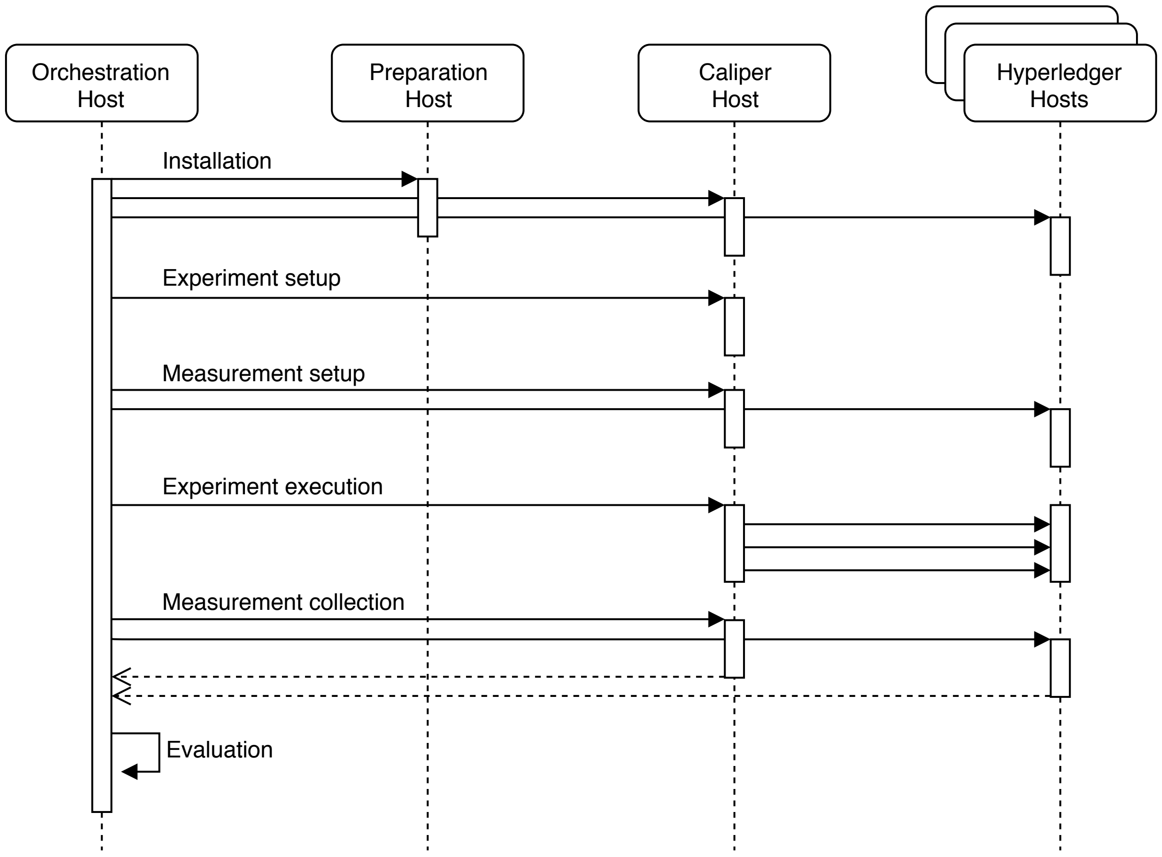

Figure 6.1: Framework Workflow

The base of the Ansible configuration is the inventory file, which maps actual machines to the groups they play in the experiment setup. There are two main groups, which are called `experiment-hosts` and `caliper`. Each node participating in the experiment should be part of the `experiment-hosts` group and a single node should be set up as `caliper`. The ledger has to be set up separately as there might exist different groups depending on the technology used. In the case of Hyperledger Fabric `fabric-peer`, `fabric-orderer`, and `fabric-ca` are defined.

Figure 6.1 shows the workflow that is executed by Ansible, resembling the design from chapter 5. The Ansible playbooks implementing these workflow steps are split into two parts. Most functionality is generic and uses the generic groups `experiment-hosts` and `caliper`, while other functionality implements the ledger specific deployment steps. The ledger playbooks are prefixed by the ledgers name (i.e. `fabric-*.yml`). The depicted installation phase begins with the generic testbed setup phase, which figure 2 is an excerpt of. It also includes the ledger specific part of installing and preparing Hyperledger Fabric. For this purpose the preparation host is used, which generates crypto material that has to be shared between the ledger hosts. While this could be done on the orchestration host itself, it requires some Hyperledger Fabric dependencies. To reduce the impact on the shared orchestration host, this functionality is outsourced to a host which is part of the testbed.

The actual deployment of Hyperledger Fabric takes place after the base setup of the testbed is done. In the Ansible inventory file the Fabric roles peer, orderer, and ca are mapped to hosts of the testbed. While in sample networks of Hyperledger Fabric the participating nodes are executed in a Docker environment, this virtualization layer is removed in this deployment and the roles are executed directly on the machines. Instead the binaries of the required Fabric version (in this thesis v1.1) are built on the nodes. Additionally to make the Fabric network work, it is necessary to configure all nodes in accordance with the testbed. This includes linking the hosts between each other and deploying crypto material which has been generated on the preparation host. Afterwards the ledger network is prepared and can be started for measurements.

6.3 Measurement Phase

Any subsequent steps are re-executed for each measurement run. This implies that the ledger is not yet configured as the specific parameters like block size might depend on the experiment. Because of this, the ledger configuration can be modified in this step. After configuration the ledger is started up, which is the last step before the measurement is started.

The measurement playbook consists of a few individual steps: First of all the monitored nodes are synchronized to the local LRZ NTP server to prevent any time offsets in measurement results. Subsequently the nodes are prepared by installing `tshark` which includes `pcapdump`. pcapdump is set up and executed as a service, producing a pcap file of all traffic visible to a node.

Just before the measurement is conducted the configuration files for the measurement are pushed to the Caliper host, and a pre-measurement script is executed. This pre-measurement script is run on all experiment hosts to apply last changes like the application of network loss.

Afterwards Caliper is executed, which is described in more detail below. As soon as the blocking execution of Caliper is finished the post-measurement script is executed on all hosts. This undoes any changes applied by the pre-measurement script and allows the following steps to proceed unimpaired.

The next step is extraction of the results. The monitoring pcapdump service is stopped on all nodes and their results are compressed and fetched into a results directory on the orchestration host. From the Caliper host the report.json, which contains all transaction data, is also fetched into the same directory.

The final phase within the testbed is the cleanup phase, which stops any running ledger processes and deletes all data created during execution. This resets the network and prepares it for subsequent measurements by removing created chains, other Hyperledger application data, and dropping docker containers that contain chaincodes and their images.

Caliper

Caliper is as previously described a benchmarking tool for blockchains. It already supports some Hyperledger blockchain implementations and is built to be easily extensible to other technologies. In this thesis this tool is utilized to define, generate and execute the workloads. Additionally it also provides the on-chain metrics for evaluation. To get an overview over Caliper a brief explanation of its architecture and workflow is given in the following:

Caliper Architecture

Caliper provides a generic benchmarking layer. It allows to define use cases mostly independent from the tested technology. The core layer provides the main modules of this tool:

- Monitoring implementations to log resource usage of local and limited remote monitoring.
- Performance analyzing, evaluating the monitored data and transaction metrics.
- Report generation, which gathers all results and generates a readable report.

In addition to this there also exists a generic blockchain interface which is utilized by the benchmarking layer to define the workloads. The blockchain interface defines the following functions:

- `init()` to initialize the ledger for the test environment
- `prepareClients()` prepares the clients with the corresponding SDK
- `getContext()` & `releaseContext()` to handle context objects defined by SDKs like Fabric's
- `installSmartContract()` to install smart contracts (chaincode in the case of Fabric) on the ledger
- `invokeSmartContract()` to run a smart contract on the ledger
- `queryState()` to read state from the ledger (in Fabric implemented by invoking but not committing a chaincode)
- `getDefaultTxStats()` to get a normalized view of the ledgers statistics

This interface is implemented in the final layer for the different technologies that are supported. The so called adaptation layer contains modules for each supported blockchain. Because of this the overall implementation is kept as generic as possible and only the lowest layer is technology-specific.

Caliper's easy extensibility allows to use it without major modifications. One change that has been made, is the extraction of all transaction data instead of single aggregated values. To allow this the client code has been modified to send the complete history from the individual client

```
 1 {
 2  "blockchain": {
 3   "type": "fabric",
 4   "config": "benchmark/simple/fabric.json"
 5  },
 6  "test": {
 7   "name": "Sample Experiment",
 8   "description" : "a short experiment with a single round",
 9   "clients": {
10    "type": "local",
11    "number": 5
12   },
13   "rounds": [{
14     "label" : "open",
15     "txNumber" : [100],
16     "rateControl" : [
17         {"type": "fixed-rate", "opts": {"tps" : 10}},
18     ],
19     "callback" : "benchmark/simple/open.js"
20    }]
21  },
22  "monitor": {
23   "type": []
24  }
25 }
```

Listing 4: Sample Caliper configuration file

processes to the main process. There the transaction history data is gathered, including individual transaction creation, endorsement, ordering, and finalization times. This data exported into a `report.json` file, in addition to the already existing HTML report file generated by Caliper.

The main interaction with Caliper takes place via two configuration files. The first is a ledger specific configuration file, which is passed to the technology specific SDK. In the case of Fabric this includes network configurations, like roles and IP addresses, and also channel, chaincode, and endorsement policy configurations. This file is referenced in the second JSON configuration file which is included exemplarily in listing 4.

The second file is split into three distinct sections. The first section defines the evaluated blockchain and references the ledger specific configuration file in line 4.

The second section defines the test and what workload is carried out during its execution. This section begins with metadata like name and description of the test in lines 7-8. It continues with the definition of the test clients, which execute the workload. In this case Caliper is instructed to spawn five clients (line 11) on the local machine (line 10). Each of these clients executes a part of the task that is defined next. The execution may consist of multiple rounds, which are executed sequentially and each has its own label (line 14). A round is defined by the amount of transactions that are created (line 15), the rate at which the transactions are generated (lines 16-18), and the workload of these transactions. Even though it is not the case for this sample configuration, Caliper also supports subrounds, which allow to execute the same workload, with different amounts of transactions and different rates.

The workload is defined by a callback entry in line 19, which is a reference to a Javascript file implementing an `init()`, `run()`, and `end()` function. These functions utilize the generic blockchain interface to allow interchangeability.

Finally the configuration also contains a section for monitoring, which is left empty for all experiments in this thesis, as this data is gathered more reliably in other ways.

Chaincodes

To simulate different applications, different chaincodes are executed in the experiments. In the following three chaincodes are described, where the first one is part of the Caliper implementation and the others have been implemented as part of this thesis.

"Simple" Chaincode

The "Simple" chaincode is a sample application that supports four operations:

- **open**: creates an account with a given balance if it does not exist yet
- **delete**: deletes an existing account
- **query**: queries the balance of an account
- **transfer**: transfers units / money between two accounts

Basically it implements a rudimentary cryptocurrency. It is a good example application with both invoked (open, delete, transfer) and queried (query) operations. To interpret the measurement results on a low level it initially makes more sense to take a look at even simpler chaincodes.

"DoNothing" Chaincode

The "DoNothing" chaincode does exactly what its name suggests. All its functions immediately return successfully. Listing 5 depicts the two required functions of chaincodes: The `Init()` function of the "DoNothing" chaincode (line 1) which is executed when the chaincode is instantiated and the `Invoke()` function (line 5) which is called each time the chaincode is invoked. Both do not expect any parameters apart from the injected `ChaincodeStubInterface` and immediately return with an empty response.

```go
func (t *DoNothingChaincode) Init(stub shim.ChaincodeStubInterface) pb.Response {
		return shim.Success(nil)
}

func (t *DoNothingChaincode) Invoke(stub shim.ChaincodeStubInterface) pb.Response {
		return shim.Success(nil)
}
```

Listing 5: “DoNothing” chaincode

“CPUHeavy” Chaincode

A slightly modified chaincode is the “CPUHeavy” chaincode. It differs from the “DoNothing” chaincode in that it executes a CPU intensive task in its invoke function before returning. This task is parameterized and accepts the amount of iterations that shall be executed. That parameter allows to experiment with the time the execution of the chaincode needs.

The actual implementation can be seen in listing 6, where the parameter is read in line 2 and the iteration takes place in lines 4 and 5.

```go
1 func (t *CpuHeavyChaincode) Invoke(stub shim.ChaincodeStubInterface) pb.Response {
2 	_, args := stub.GetFunctionAndParameters()
3 	iterations, _ := strconv.Atoi(args[0])
4 	for i := 0; i < iterations; i++ {
5 	}
6 	return shim.Success(nil)
7 }
```

Listing 6: “CPUHeavy” chaincode excerpt

“DataHeavy” Chaincode

The “DataHeavy” Chaincode aims to put load on the network while relieving the ledger. Code-wise the chaincode’s implementation is identical to “DoNothing”. But during execution the client is appending a parameter of predefined size. Even though its content is not used, the data is still transfered through the network.

Listing 7 shows the implementation of the dataHeavy workload utilizing Caliper’s API. The init method accepts arguments which include a `bytes` parameter. This parameter defines the size of the `Int8Array` in line 5 of listing 7. Everytime the workload is run, the array is passed as a parameter (line 10).

```
 1  let bc, contx, payload;
 2  module.exports.init = function(blockchain, context, args) {
 3      bc       = blockchain;
 4      contx    = context;
 5      payload  = new Int8Array(args.bytes);
 6      return Promise.resolve();
 7  };
 8
 9  module.exports.run = function() {
10      return bc.invokeSmartContract(contx, 'dataHeavy', 'v0', {verb:
     ↪   'dataHeavy', payload: payload}, 30);
11  };
```

Listing 7: “DataHeavy” workload

6.4 Evaluation Phase

The evaluation is written in Python 3 and is completely independent from the previous steps. All measurement results are persisted within a single directory on the orchestration host. Thus the evaluation can take place even after other measurements have happened and also on a different machine outside of the experiment network.

Preprocessing

The evaluation phase begins with a preprocessing step. The goal of this step is that the pcap dump shall be unified to a single file, that contains all relevant traffic in the network. Therefore the first step is deduplicating the data and removing background traffic. All pcap files are filtered to only contain outgoing TCP messages. Filtering only for TCP messages works with Hyperledger Fabric, as most of the communication is done through the gRPC protocol, which runs on top of TCP. Incoming messages are removed such that in the next step, where the pcap files are merged, the packets are not added twice.

Both filtering and merging is done with the help of the libtrace[1] tool. During implementation with libtrace two issues occurred: The Python bindings of libtrace[2] initially did not work as

[1] https://research.wand.net.nz/software/libtrace.php

[2] https://github.com/nevil-brownlee/python-libtrace

expected and the debugging is difficult, because the error handling is lacking. Error messages generated in the C code of libtrace are not forwarded by the python bindings. Thus if an error occurs it is difficult to identify its origin. Because of this the command line tools are called directly from Python via `subprocess`. Secondly the libtrace library does not support writing compressed pcap files. Therefore compression is removed during this preprocessing step. After these steps have finished there exists a new `merged.pcap` file within the results directory.

Reporting

The main evaluation takes place in a Jupyter[1] notebook, which reads the `merged.pcap` and `report.json` files, transforms their data and visualizes it. Tools utilized for this task include:

- **dpkt**[2] which allows simple parsing of pcap files
- **pandas**[3] which offers data structures and data analysis tools, allowing simple transformation of data, based on the R language
- **seaborn**[4], a graphics tool based on matplotlib, allowing to visualize statistical data easily.

Initially the network data is read in and for each TCP packet its send time, source, destination and ethernet frame size is stored in an array. Source and destination IP address are already replaced by a descriptive name for the node (e.g. caliper, peer0.org1). Afterwards this array is converted into a dataframe, which is the base data structure of pandas.

Pandas functions are utilized to process and transform the data. This includes converting the absolute timestamps to relative times from the experiment's start time and resampling the data to a fixed frequency, grouping and aggregating all entries falling into a single bin. In combination with pandas transformation operations like pivoting and unstacking, this allows to generate interesting insights into the gathered data.

A similar procedure takes place for the data extracted from the Caliper report file `report.json` that contains the transaction metadata. Timestamps for transaction creation, ordering, and finalization are extracted, processed, and transformed.

The prepared data is then visualized through seaborn in multiple views. Graphics are immediately displayed in the Jupyter notebook to verify the results and are also stored in the results directory for future usage.

[1] https://jupyter.org/

[2] https://github.com/kbandla/dpkt

[3] https://pandas.pydata.org/

[4] https://seaborn.pydata.org/

Chapter 7

Evaluation

In this chapter a few exemplary experiments are described to show the capabilities of the framework in its current state. It also gives a few insights into the performance of Hyperledger Fabric.

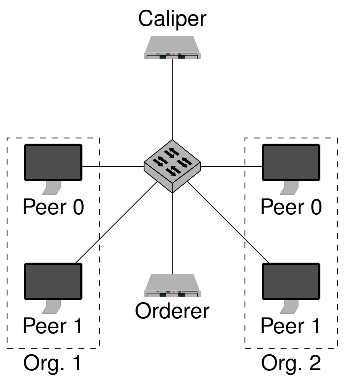

Figure 7.1: Testbed network structure

Six hosts of the testbed are set up as depicted in figure 7.1. A use case where two organizations participate is simulated. Each organization provides two ledger peers to the network which maintain a copy of the ledger. A single orderer host is responsible for the creation of blocks and the Caliper host executes the workloads. These six hosts are part of a star topology and carry out the following measurements.

7.1 Introductory experiment

The following shall give an understanding of Hyperledger Fabric's function and the results the framework returns. Therefore a basic experiment has been conducted and is described below. The measurement shows the results of the execution of the configuration file, described in listing 4 of section 6.3. It executes a single round with 100 transactions at a fixed rate of 10 transactions per second (table 7.1).

Rounds:	1
Transactions:	100
Rate:	10 tps
Varied Factor:	–

Table 7.1: Introductory measurement

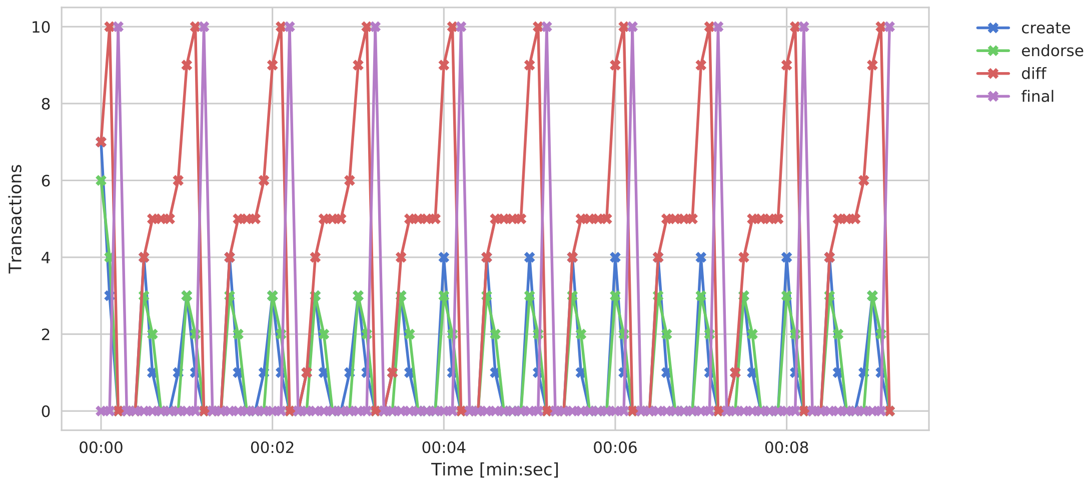

Figure 7.2: Sample experiment transaction timings

On the application layer, where the interaction with the ledger is defined by transactions, the transaction timings give insight into the performance. Figure 7.2 depicts the transaction timings gathered by Caliper in four curves. During preprocessing the data visualized in this graphic is normalized into a frequency of 100 ms. The individual data points are marked in this graphic while they won't be highlighted individually in the later graphs.

The "create" and "final" curves indicate when transactions are created and finalized. Transaction creation is happening periodically as defined in the Caliper configuration. This means 10 transactions every second in batches of 5 because every client process of Caliper generates transactions. Finalization takes place when the transactions have traversed through the complete ledger.

The "endorse" curve is specific to Hyperledger Fabric. Transaction endorsement is done by the peers that are configured to be endorsers immediately after transaction creation and happens before ordering. This is the reason why the creation and endorsement curves only differ slightly. Most of the endorsements take place in the same measurement timestep as the transaction creation, but some return in the next timestep.

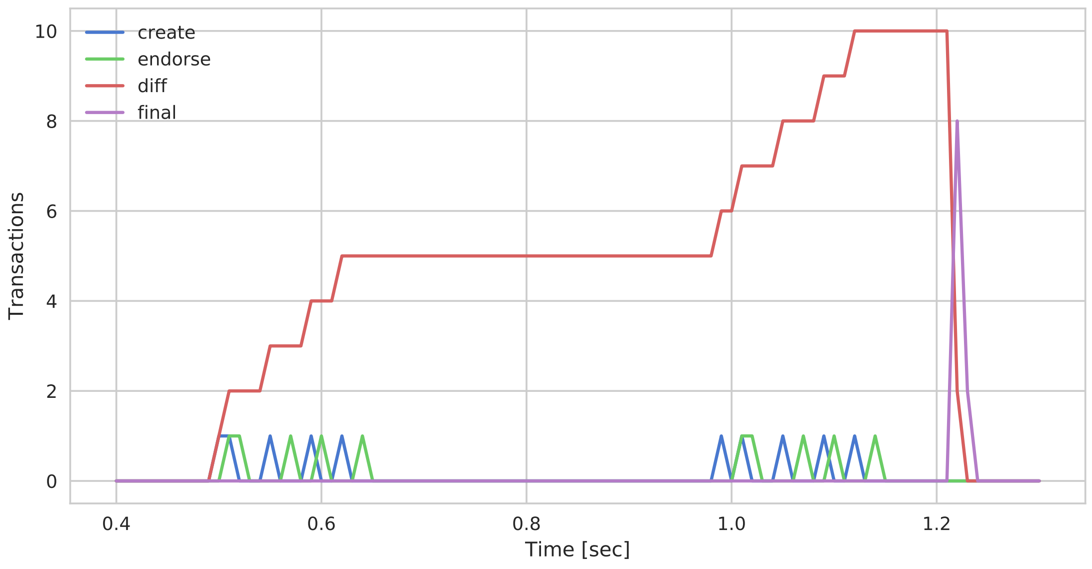

Figure 7.3: Sample experiment transaction timings (excerpt)

Finally the "diff" curve shows the difference between the cumulative amount of finalized and created transactions, which represents the backlog of transactions. The backlog grows with each transaction creation and falls off when a new block is created and the transactions are finalized. As the finalization depends on a block the finalization happens in batches of 10 transactions, which is the configured default block size. Overall there are ten blocks created during this measurement.

Figure 7.3 shows a zoomed version of figure 7.2. It depicts the second block creation phase from 400 to 1300 ms of the same measurement with a sample rate of 10 ms. The reduced sample rate allows to follow each individual transaction creation and endorsement. Again it is visible that at about 500 ms five transactions are created and soon after are endorsed. Each of the five clients creates one transaction and communicates with the endorsing peers to endorse it. Even though it is not depicted in this graph the endorsed transactions are forwarded to the orderer immediately. But as the default block size is set to 10 transactions no transactions are immediately finalized. Instead after a short delay each client creates another transaction. After their endorsement they are forwarded to the orderer which is now able to generate a block. This block is published to all peers in the network and the clients and Caliper are notified of the finalization of the transactions. While it would be expected that the finalization curve spikes up to ten transactions, the transactions are not all finalized in the same 10 ms. The first transaction is finalized at 1224 ms into the measurement, while the last transaction is acknowledged at 1232 ms which falls into, a second sampling timestep. Thus figure 7.3 depicts a spike to 8 transactions at 1220 ms and another 2 transactions in the following timestep.

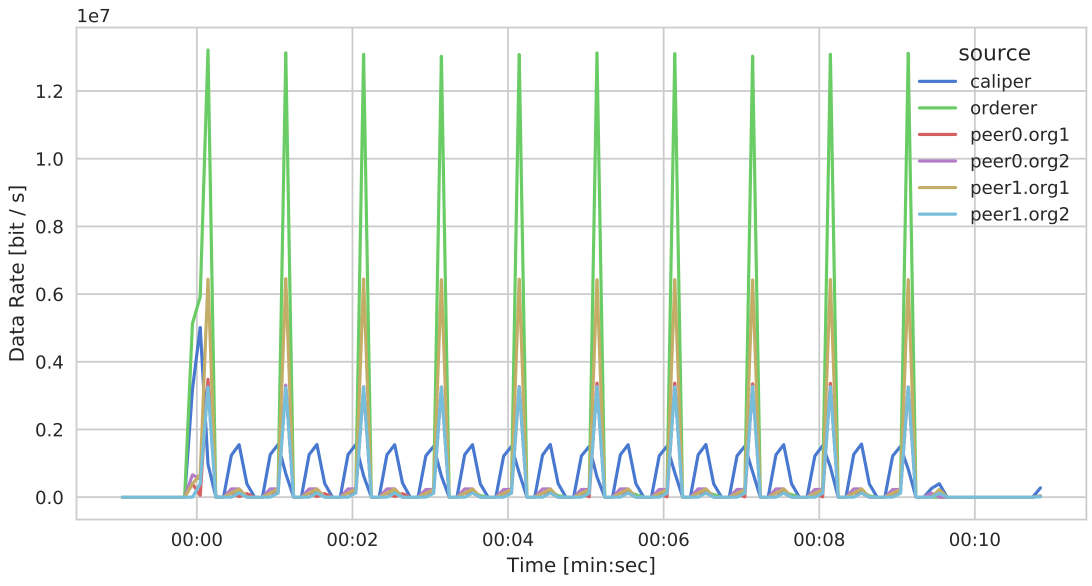

Figure 7.4: Sample experiment data rate

The actual communication between the peers is shown in figure 7.4. It depicts the outgoing traffic rate from all nodes in the network in bits per second. The node sending the most traffic is the Caliper node. Traffic originating from this node corresponds to the create and endorse transaction timings. When transactions are created Caliper sends them to some peers which respond with the endorsement. The first transactions are just forwarded to the orderer, but after ten transactions the orderer creates a block and forwards it to all peers in the network. Broadcasting of this block explains the reoccurring spike of the orderer's data rate. Traffic of the peers also rises because they notify Caliper of the acknowledged transactions contained in the block they just received.

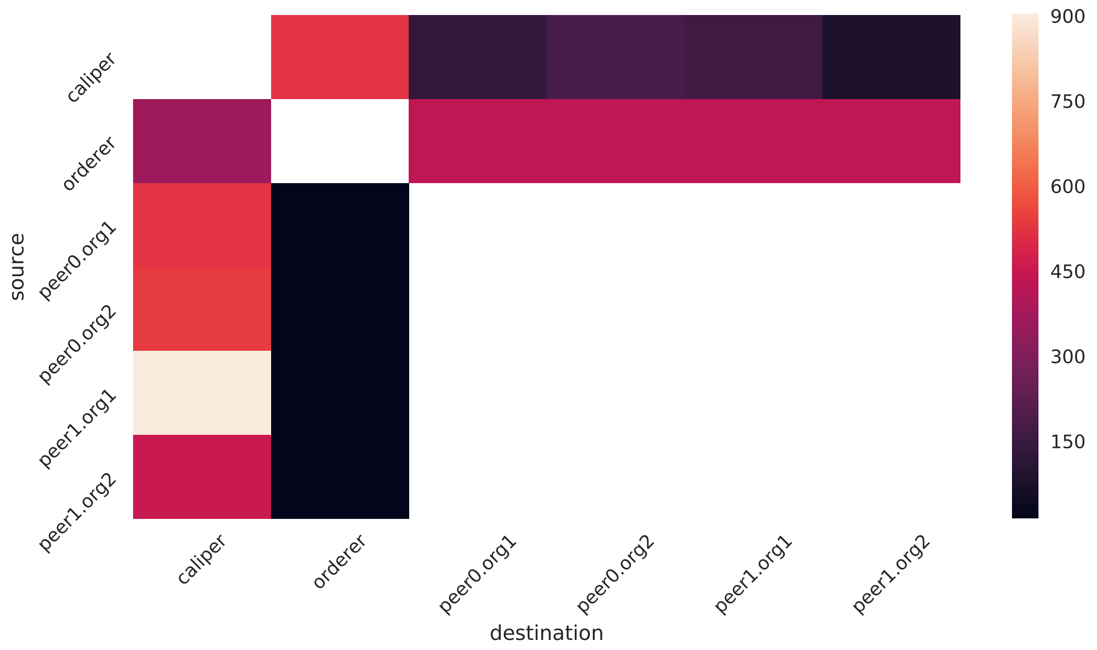

Figure 7.5: Sample experiment heatmap of network traffic in KB

Figure 7.5 represents a heatmap of the communication between the nodes in the network. The white fields including the diagonal and all inter-peer communication represent no traffic. Otherwise there is minimal communication from the peers to the orderer, in total about 54 KB. Also Caliper does not send large amounts of data to the peers, even though the amount of messages is comparably high. While there is significant traffic originating from the orderer, the most data is transmitted from the peers to Caliper. The reason that the Peer 1 of Org 1 has outgoing traffic of 901 KB, while the other peers have around 500 KB of outgoing traffic, is that the Fabric SDK chooses any peers to endorse transaction. Repeated executions of this measurement lead to different traffic distributions for the peers.

The following experiments will each focus on a specific factor and measure its effect on the performance of the DLT.

7.2 Experiment with varying transaction rate

The first interesting experiment is to observe the behaviour of a DLT under different loads on the application layer. This experiment executes a simple wallet transaction at different rates while keeping the total amount of transactions constant. Specifically this is the workload defined in section 6.3. The different rates are executed in multiple rounds within the same measurement. Rounds are separated by 5 second pauses from each other by Caliper. Beginning with 100 transactions per second (tps), the rate is increased by 100 tps in each round over a total of ten rounds. Each round runs until 1000 transactions have been sent (table 7.2).

Rounds:	10
Transactions:	1000 / round
Rate:	from 100 to 1000 tps
Varied Factor:	transaction rate

Table 7.2: Measurement with varying transaction rate

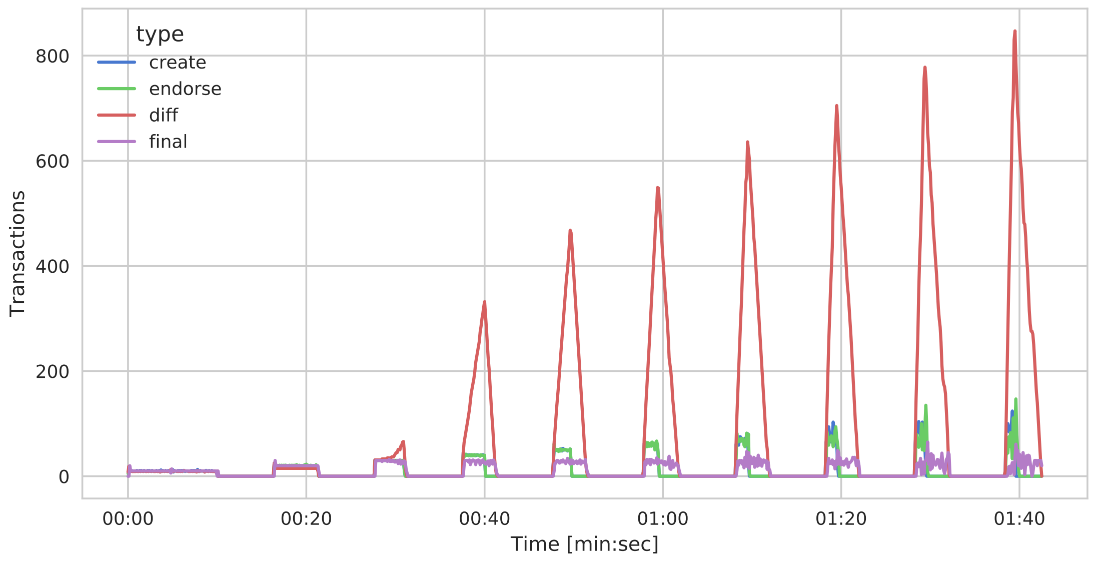

Figure 7.6: Transaction timings with varying transaction rate

Evaluating the transaction timings in figure 7.6 gives a few first insights. For the first two rounds with 100 and 200 tps there is just a minimal offset between the create and final curve, whereas beginning with 300 tps the curves separate. Especially from the fourth round onwards it is visible, that the “final” curve separates from the other two. For all rounds upwards of 300 tps the tps increases and thus the transactions are created in a shorter time, while the rate at which transactions are finalized stays mostly the same. This separation leads to a backlog of transactions. Any transaction that is in or waiting for execution contributes to the backlog until the client is notified of the transaction’s successful commit to the ledger.

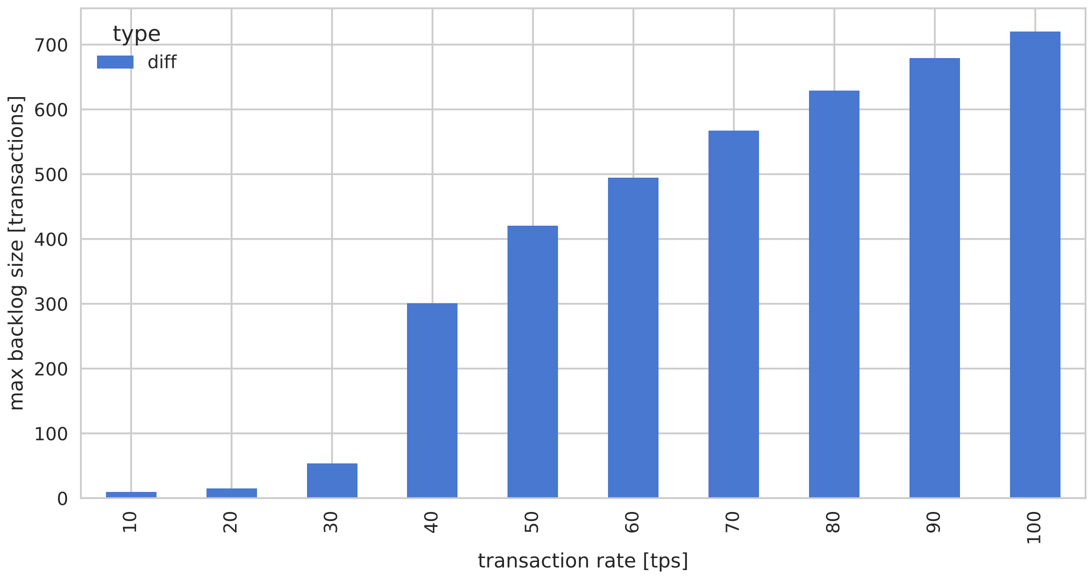

Figure 7.7: Transaction backlog with varying transaction rate

While the backlog grows only slowly initially in the 300 tps round, the amount of transactions in the backlog increases immensely in the following rounds. The newly created transactions cannot be processed in time, hence the backlog grows until all 1000 transactions are created. This development of the backlog is summarized in figure 7.7. Displaying the maximum backlog size in each round, it shows that the first two rounds have a small constant backlog. Beginning with the third round the backlog size increases with the transaction creation rate.

The increase in backlog size has a direct impact on the transaction latencies as depicted in figure 7.8. A small backlog in the first rounds allows low latencies of 100 ms, the increasing backlog size leads to delayed transaction timings. While the average transaction latency increases, the variance increases, too. Even in the last round the first few transactions have a low latency because there is no backlog that could impede them yet. So within a round the transaction latency as well as the backlog size increases.

This behaviour can also be spotted in the evaluation of the network data (figure 7.9). The outgoing network traffic originating from the Caliper node corresponds mostly to the creation of transactions, whereas the orderer and peers participate in the execution of the transactions. Similar to the “create” curve in figure 7.6 the outgoing traffic of the caliper node drops before the other nodes in the network are finished. This offset again is especially visible in the fourth round where the transaction creation rate is 400 tps.

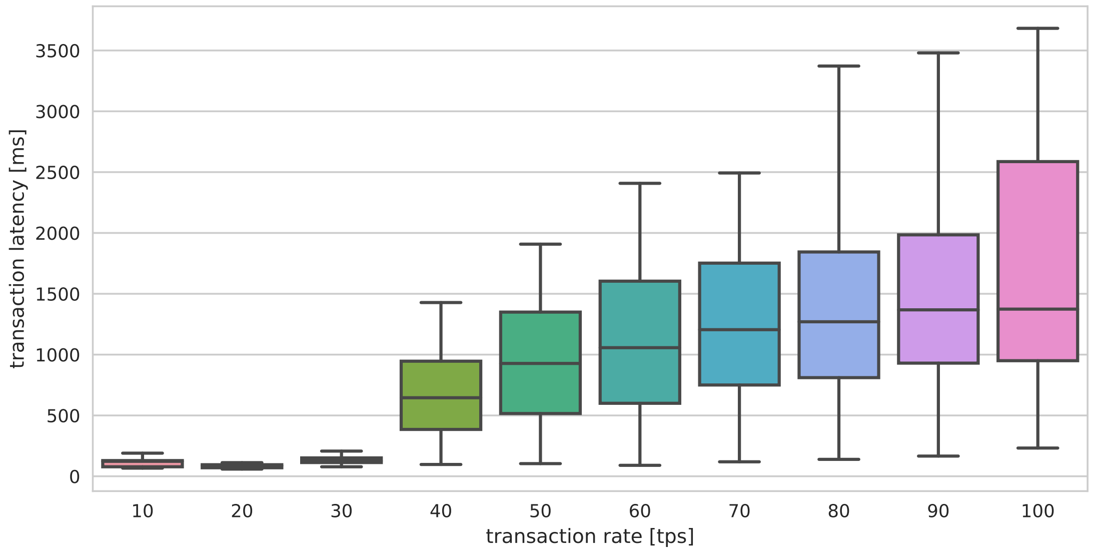

Figure 7.8: Transaction latencies with varying transaction rate

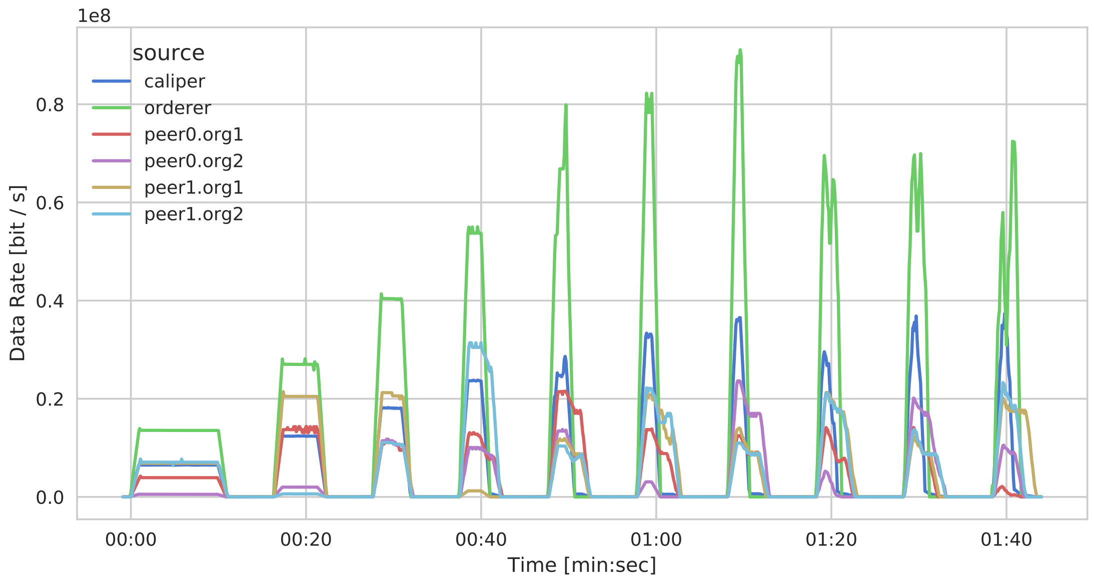

Figure 7.9: Data rate with varying transaction rate

To inspect this in more detail, there is a second measurement focusing on low transaction creation rates. This measurement consists of 15 rounds with again 1000 transactions each. Transaction creation rate reaches from 20 tps in the first round to 300 tps in the last round (table 7.3).

Rounds:	15
Transactions:	1000 / round
Rate:	from 20 to 300 tps
Varied Factor:	transaction rate

Table 7.3: Measurement with low varying transaction rate

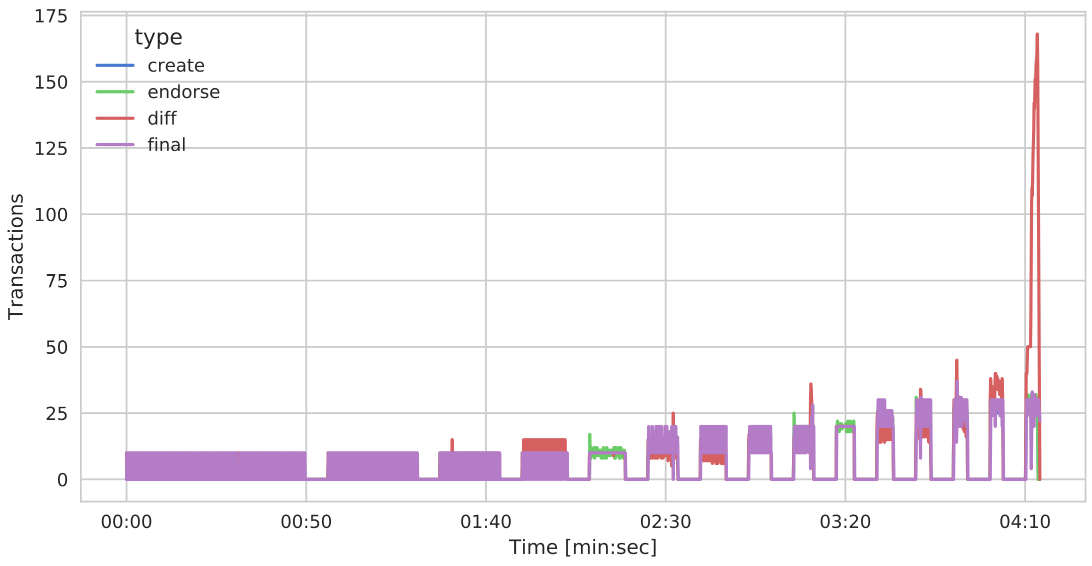

Figure 7.10: Transaction timings with low transaction rate

Figure 7.10 shows the results of this measurement in regard to the transaction timings. Except for a few outliers the backlog begins to build up in the second-to-last round with 280 tps and jumps up even further in the last round. This is also visible in figure 7.11 which summarizes the backlog size by taking the 90% quantiles of each round's backlog size. Using the quantile instead of the maximum removes any outliers, because of momentary backlog jumps. While this also lessens the maximum backlog size of the last round, the main outcome is preserved.

Even though all rounds except the last two show a low backlog size, the transaction latencies differ distinctly. Figure 7.12 displays these high and fluctuating transaction latencies in the first few rounds. They originate from the low transaction creation rate which is also visible in figure 7.10, where the "create" curve jumps up and down. This means that there are timesteps in which there are no new transactions created. In the case of Hyperledger Fabric this leads to blocks that are not immediately filled up and thus delaying the first few transactions within a block. As there are more transactions created per second this waiting time per block is reduced or even completely removed when the orderer has a backlog of transactions to put into the next block.

In summary there is an optimal spot in regard to transactions per second, which minimizes both the backlog size and the block creation delay. In this measurement the 240 tps round gives the best results.

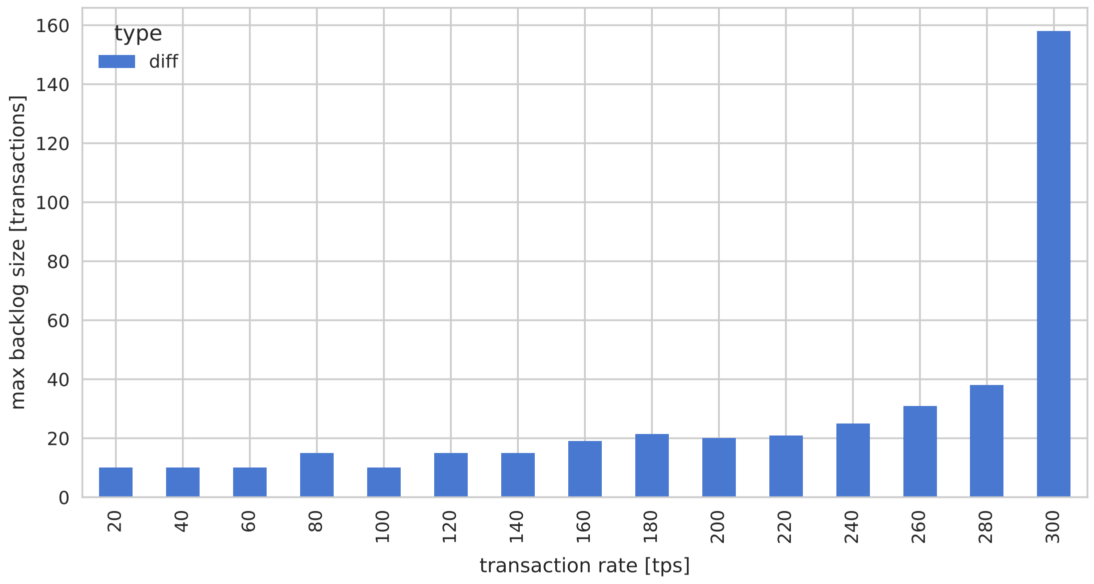

Figure 7.11: Transaction backlog with low transaction rate

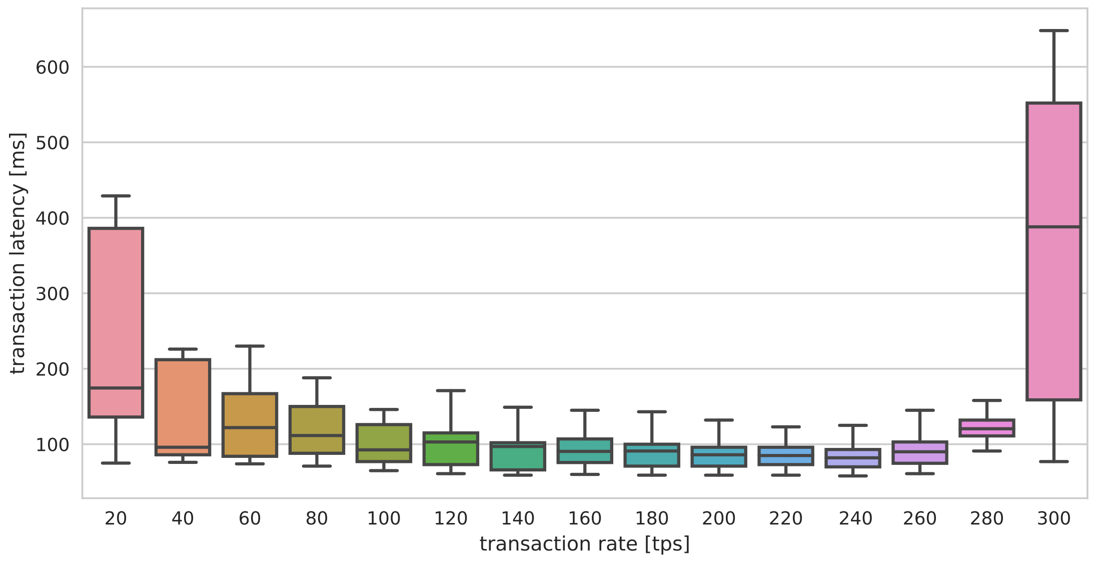

Figure 7.12: Transaction latencies with low transaction rate

7.3 Experiment with varying chaincodes

The second experiment compares the performance of different chaincodes. Previously described workloads DoNothing, CPUHeavy, and DataHeavy are executed in the same environment.

DoNothing is executed as the base case. Similar to the previous experiment, multiple rounds with increasing transaction rate are executed. Beginning with 100 tps the rate is increased to 1000 tps over ten steps (table 7.4).

Rounds:	10
Transactions:	1000 / round
Rate:	from 100 to 1000 tps
Varied Factor:	chaincode

Table 7.4: Measurement with varying chaincode

Figure 7.13 shows the data rate during the execution of the experiment with the “DoNothing” chaincode. This behaviour is similar to the simple chaincode from the previous experiment. As seen in figure 7.14 the 400 tps round is the first round that has an increasing backlog. Comparing this to the results of the “DataHeavy” workload shows an increase of the data rate (figure 7.15). The workload in this experiment is configured to pass 10 KB as a parameter to the chaincode on the peers. While the duration of the rounds stays mostly the same, the peers in the network have to forward more data. Thus the outgoing data of the orderer increases in the first round (100 tps) from about 1.5 Mbit/s for the “DoNothing” chaincode to just below 5 Mbit/s for the “DataHeavy” chaincode. Figure 7.16 shows that the transaction timings hardly differ between the two measurements. Mainly only the third round with 300 tps is different in that there is a small build up of a backlog for the “DataHeavy” chaincode, while there is none with the “DoNothing” payload.

Finally comparing this to the measurement of the “CPUHeavy” chaincode leads to similar results. The measurement is configured to run the “CPUHeavy” chaincode with 14,000,000 iterations. On the testbed’s machines the execution of a Golang program containing just this for-loop takes 0.01 seconds on a testbed node with a CPU utilization of 100%. As expected the increased load creates a similar behaviour as the “DataHeavy” workload compared to the “DoNothing” workload. Figure 7.17 and figure 7.18 show the results of the CPUHeavy measurement with the given parameters. The transaction timings show even higher backlog growth than in the case of the “DataHeavy” workload, while the bandwidth usage stays a lot lower.

In addition to this the increased load on the nodes leads to higher transaction latencies. Figure 7.19 shows the latencies of the “DoNothing” workload, whereas figure 7.21 shows the “CPUHeavy” workload’s latencies. “CPUHeavy” overall at least doubles the average transaction time. In the last round the increased load on the nodes leads to an average transaction latency of 3.5 seconds, while the latency decreases to 1.6 seconds if there is no load. While the increase of the mean transaction latency is not as distinct for the “DataHeavy” workload, it still shows an earlier increase in figure 7.20. The transaction latencies already start to increase in the third round with a transaction rate of 300 tps, whereas the latencies of the “DoNothing” measurement have their minimum at this transaction rate.

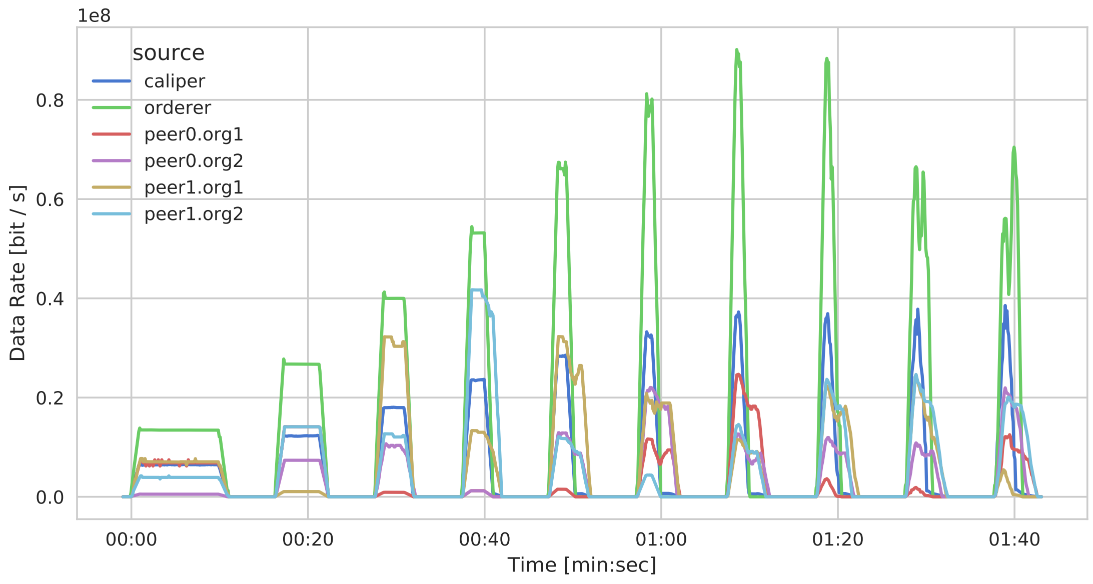

Figure 7.13: Data rate with “DoNothing” chaincode

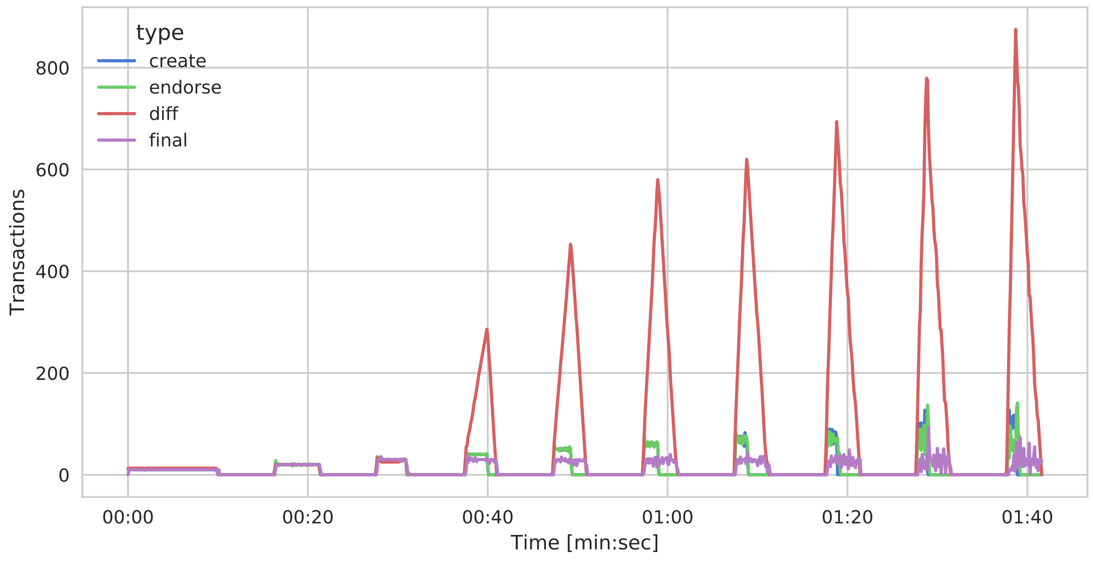

Figure 7.14: Transaction timings with “DoNothing” chaincode

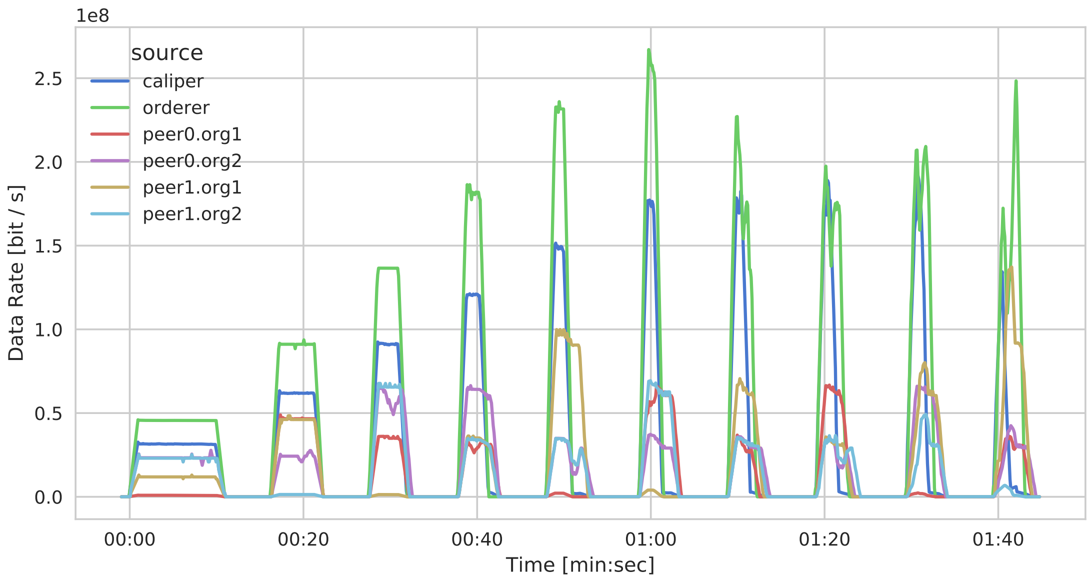

Figure 7.15: Data rate with “DataHeavy” chaincode

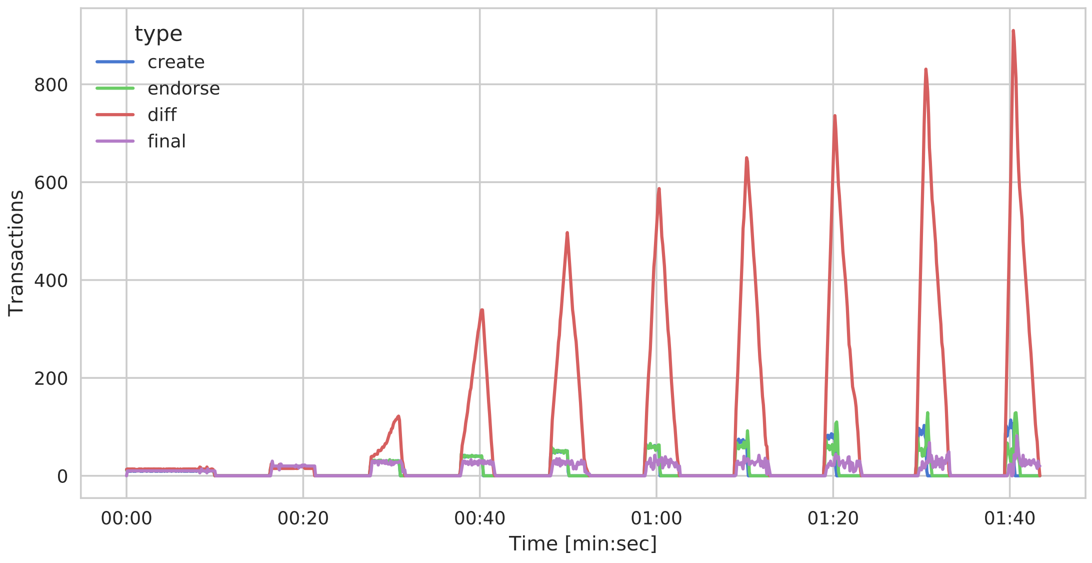

Figure 7.16: Transaction timings with “DataHeavy” chaincode

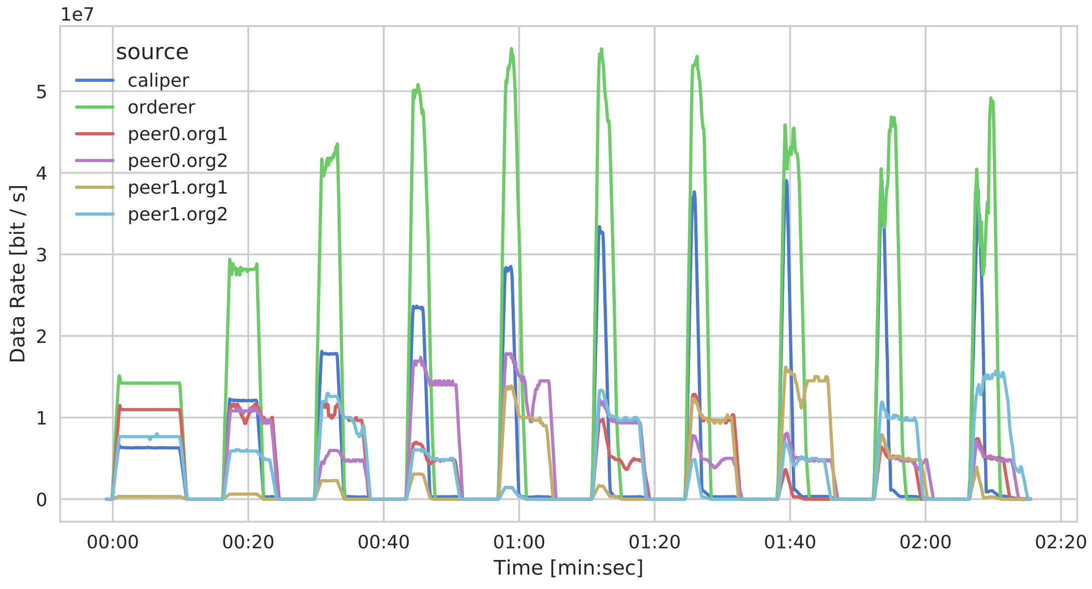

Figure 7.17: Data rate with "CPUHeavy" chaincode

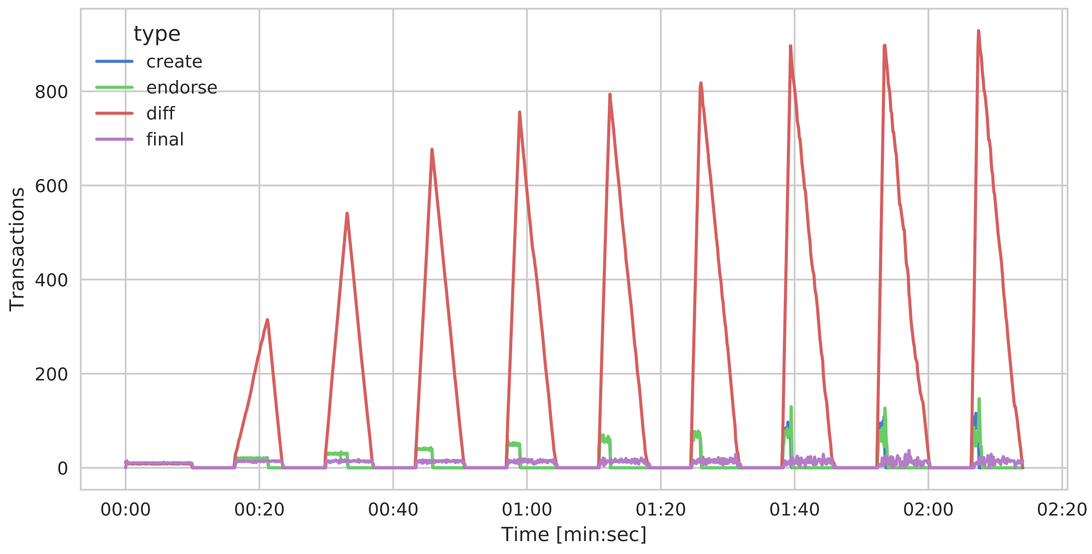

Figure 7.18: Transaction timings with "CPUHeavy" chaincode

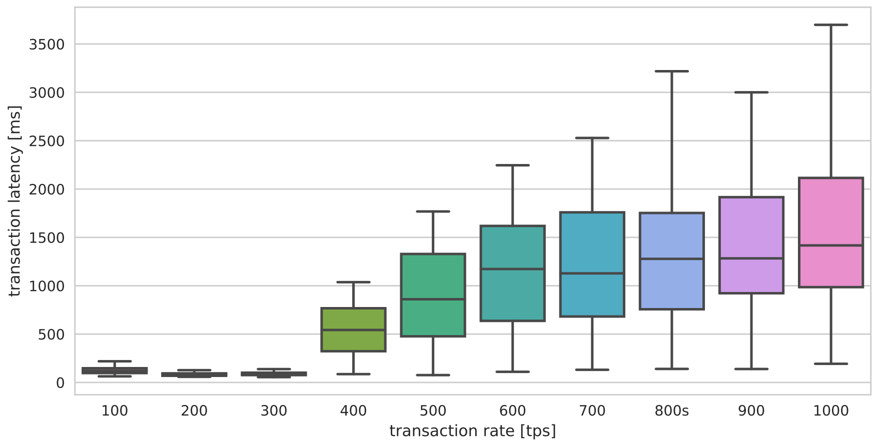

Figure 7.19: Transaction latencies with "DoNothing" chaincode

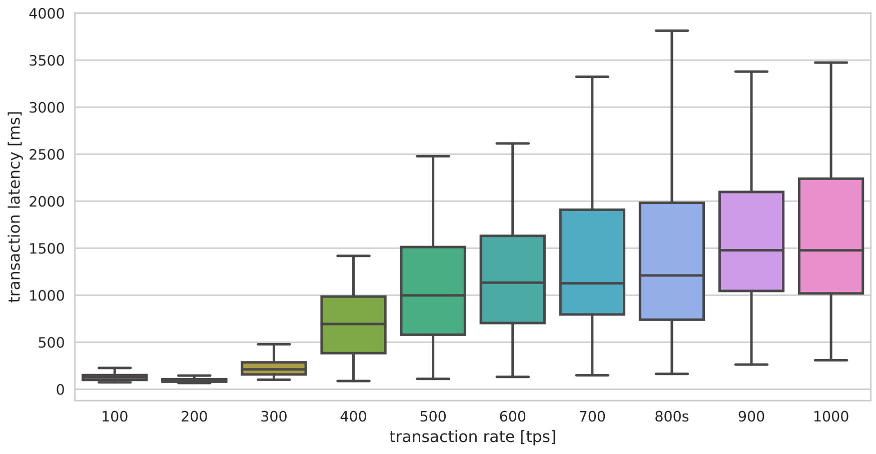

Figure 7.20: Transaction latencies with "DataHeavy" chaincode

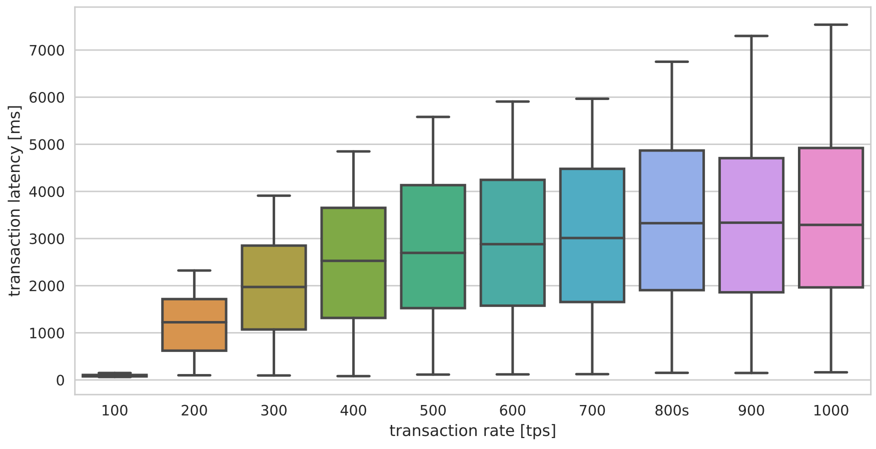

Figure 7.21: Transaction latencies with "CPUHeavy" chaincode

7.4 Experiment with varying block sizes

As mentioned in section 2.2 Fabric is built in a way, that components are easily exchangeable and configurable. One main configuration option that has great influence on the functionality of the ledger is the block size, which is configured per channel. Orderers apply this value during the batching process. They either return after a defined timeout or after it received the configured amount of messages. By default the block size is 10 transactions, but it can easily be modified. For this experiment the default value is compared with the reduced block size of 5 transactions.

Rounds:	10
Transactions:	1000 / round
Rate:	from 100 to 1000 tps
Varied Factor:	block size

Table 7.5: Measurement with varying block size

Figures 7.22 and 7.23 show the transaction timings for the measurements with default and reduced block size respectively. The measurements are similar to the previous experiments and both execute 10 rounds with 1000 transactions each, utilizing the same workload as the first experiment. These two measurements were executed with increasing transaction rates from 100 to 1000 tps (table 7.5).

The comparison of the backlog development between these two measurements shows that a lower block size in figure 7.23 leads to quicker backlog build up than with default block size in figure 7.23. In the measurement with reduced block size a backlog increase already occurs in the round with 200 tps whereas in the default setup it builds up slower and one round later. While the curve of transaction finalization stretches longer in the measurement rounds with block size 5, the endorsement curve is similar to the other measurement. Similarity in endorsement suggests, that the impact on the performance stems from the ordering phase. Specifically this performance reduction occurs because smaller blocks create an overhead for the orderer, as twice the amount of blocks has to be created. Thus at a given transaction rate the network is not able to produce and publish blocks fast enough to keep up with the creation of new transactions.

Block size also affects the transaction latencies as seen in figure 7.24 and figure 7.25. Similar to a workload increase the latencies in later rounds are significantly higher when the block size is reduced. With a block size of 5 the transaction latencies begin to increase already in the 200 tps round and in the last rounds the latency is doubled in comparison to the measurement with a block size of 10.

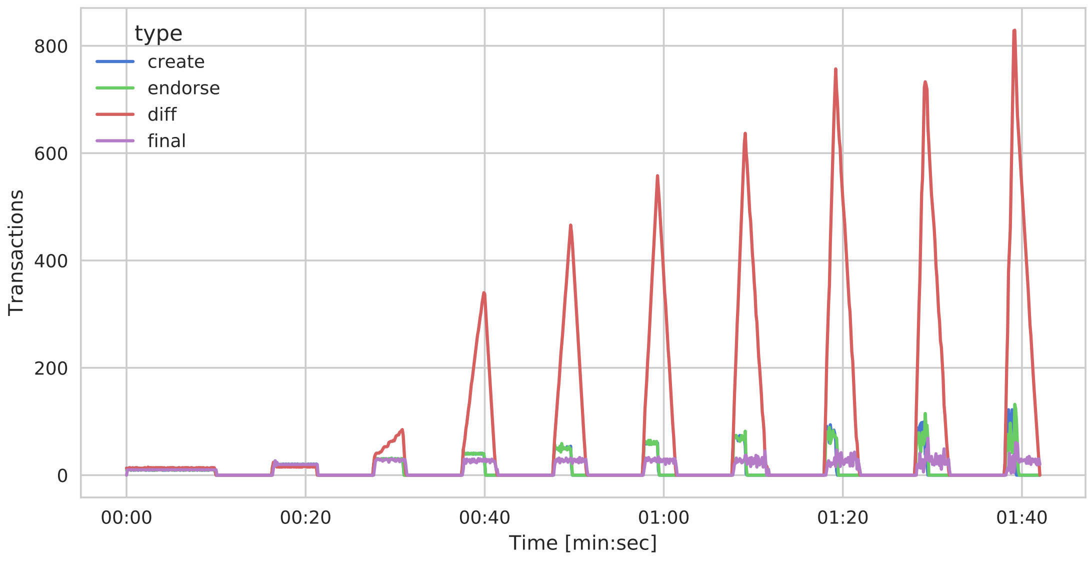

Figure 7.22: Transaction timings with block size 10

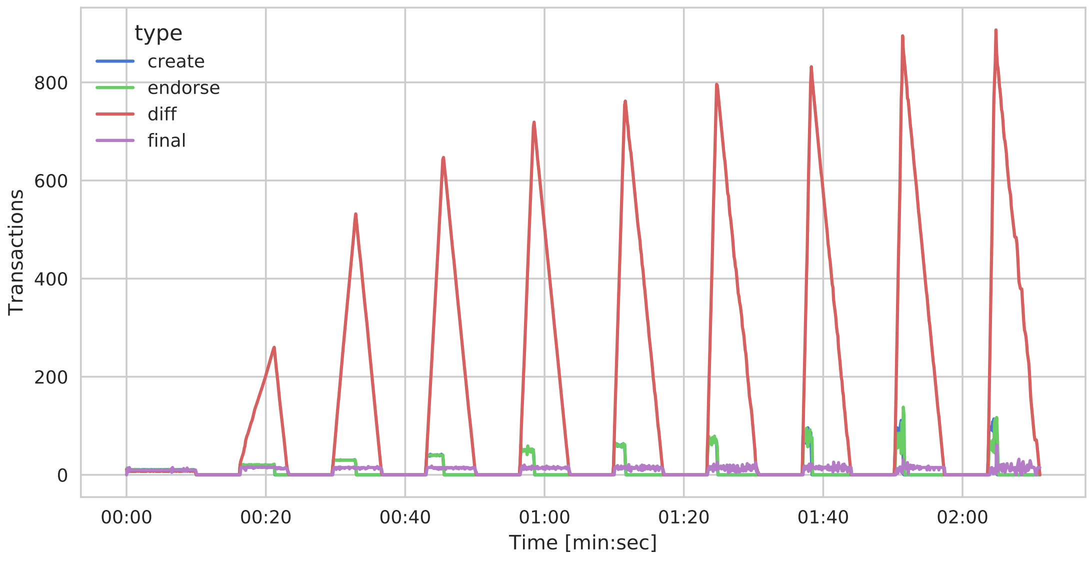

Figure 7.23: Transaction timings with block size 5

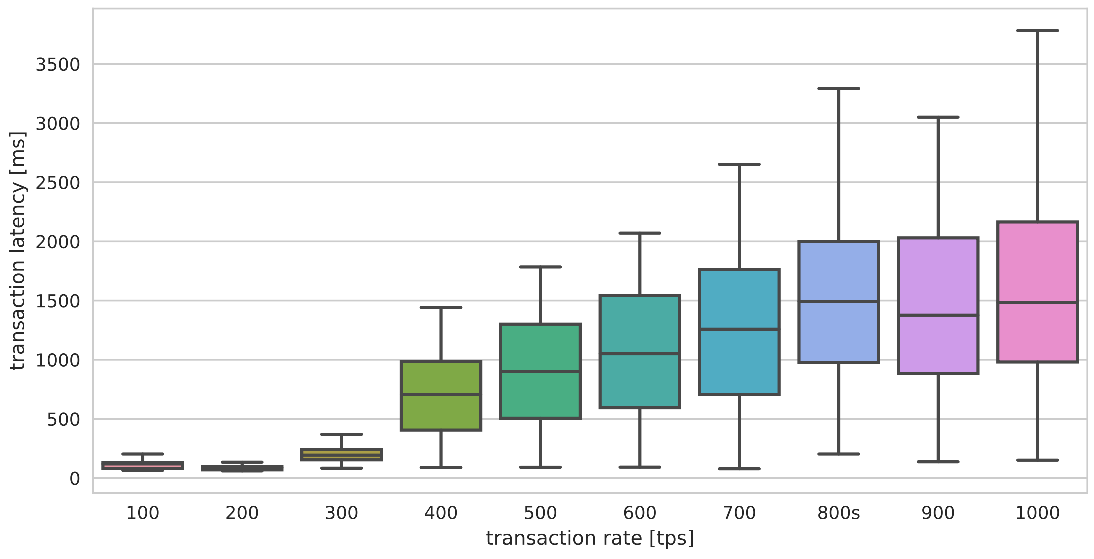

Figure 7.24: Transaction latencies with block size 10

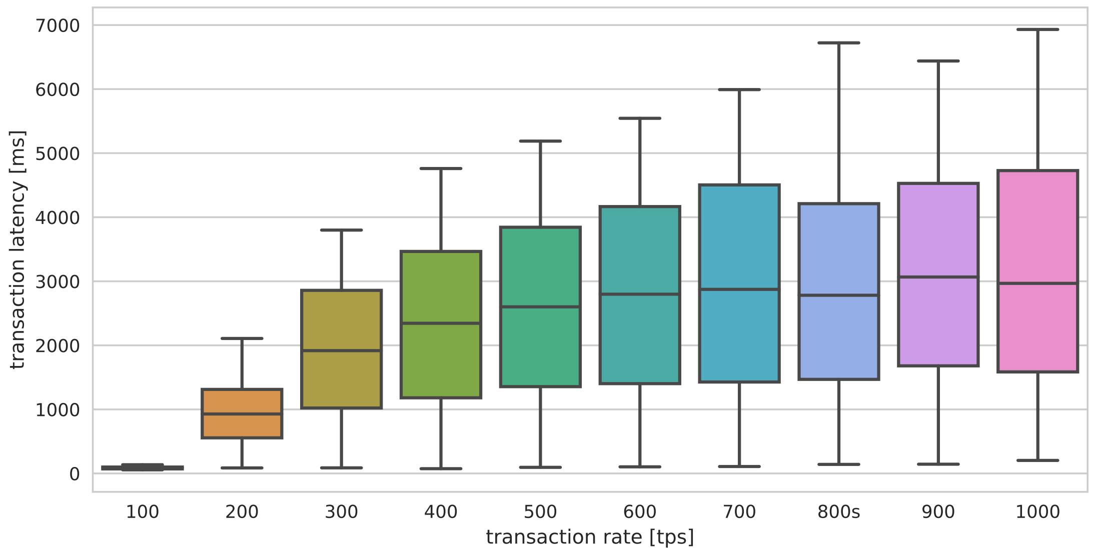

Figure 7.25: Transaction latencies with block size 5

Different results occur, when lower transaction rates are considered. Another set of measurements is conducted with transaction rates ranging from 10 to 100 tps (table 7.6).

Rounds:	10
Transactions:	1000 / round
Rate:	from 10 to 100 tps
Varied Factor:	block size

Table 7.6: Measurement with varying block size and low transaction rate

As expected for such low transaction rates there is no backlog build up, because the transactions can be processed in time. As opposed to the previous measurements this outcome is favorable towards the lower block size. Figures 7.26 and 7.27 illustrate the transaction latencies under low transaction rates. As explained in section 7.2, extremely low transaction rates cause increased transaction latencies, because the transactions can not be immediately included in a block. The block is only published when the number of transactions matches the block size or a timeout is reached. This means that at a transaction rate of 10 tps the blocks with a block size of 10 are filled up slower than with a block size of 5. Transactions are handled by the orderer in shorter intervals, if a smaller block size is configured. Because of this, figure 7.26 shows a distinctly higher transaction latency for 10 tps, than figure 7.27. This difference between the transaction latencies is reduced with increasing transaction rates and mostly disappears at the transaction rate of 100 tps.

In summary it can be recognized, that the block size factor is a configuration parameter that allows tuning of the ledger's performance. Depending on the application and its transaction rate different block sizes might be preferable. While an application that induces a high transaction rate on the network best suits a high block size, low transaction rates are supported by low block sizes.

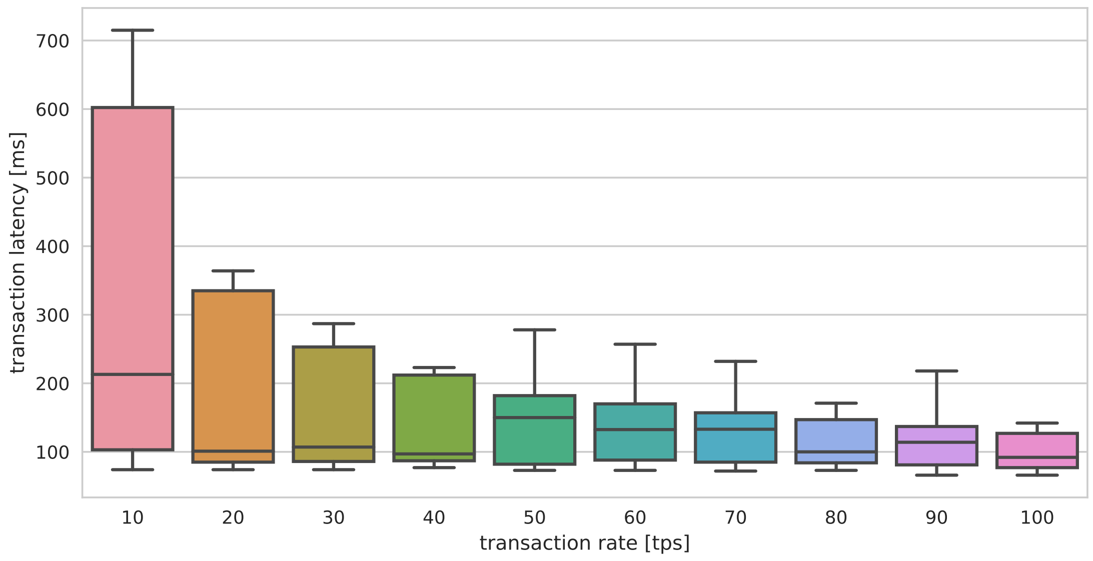

Figure 7.26: Transaction latencies with block size 10 for low transaction rates

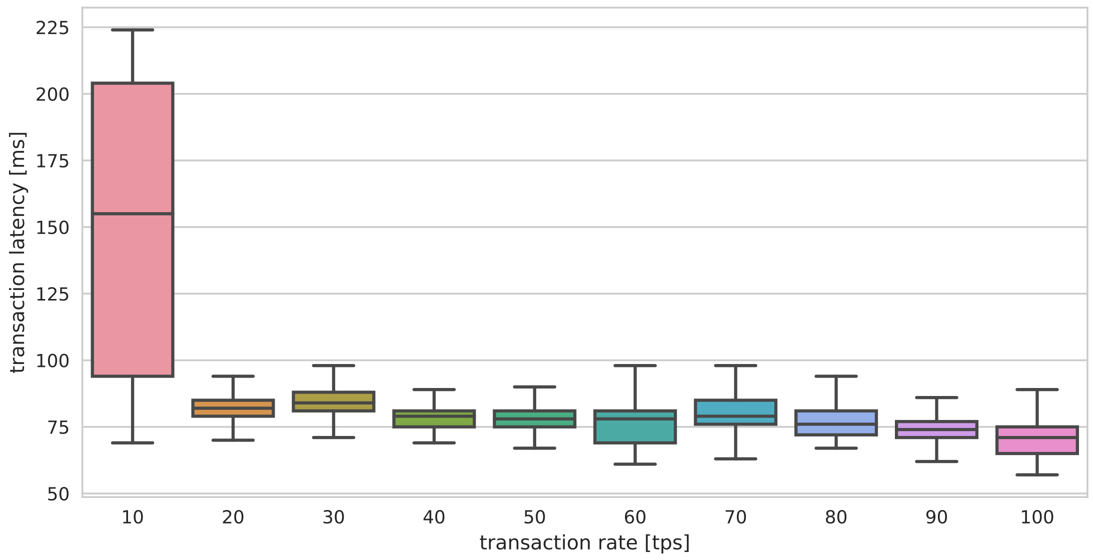

Figure 7.27: Transaction latencies with block size 5 for low transaction rates

7.5 Experiment with network loss

It is expected that network characteristics influence the ledger performance. The impact of network loss on the performance of Hyperledger Fabric is evaluated in this experiment. The experiment consists of multiple measurements with 10 rounds each. Every round executes 1000 transactions at a rate of 100 tps (table 7.7).

Rounds:	10
Transactions:	1000 / round
Rate:	100 tps
Varied Factor:	network loss

Table 7.7: Measurement with varying network loss

Three measurements are conducted: The first one is executed without any modifications, while the second measurement is conducted with a 2% network loss on all interfaces in the network. The final measurement is run under the impact of a 5% network loss. In figure 7.28 the transaction latencies of the first measurement show the same characteristics as the 100 tps round in the first experiment as it is executed under the same circumstances. The other measurements with network loss lead to increased transaction times and even failed transactions. Figure 7.29 and figure 7.30 show the corresponding latencies for the same setup with 2% and 5% network loss respectively. The increase in network loss leads to an increase in transaction latency from a mean value of 103 ms without network loss, to a mean of 134 ms with 2% network loss. For all measurements the minimum stays the same because there always are transactions that are not affected by the network loss by chance. In the measurement with 5% loss even higher latencies can be observed with a mean of 432 ms. In this measurement there also occurred extremely high transaction latency values. These outliers have been filtered out in the graphics, as they stem from failed transactions which could not be completed. The framework is configured to wait for transaction completions for 30,000 ms, so any dropped transactions timeout after this time. Over the 10 rounds with a total of 10,000 transactions 57 such errors occurred.

Figure 7.31 gives an aggregated overview of the three measurements, depicting the increase of transaction latency across them.

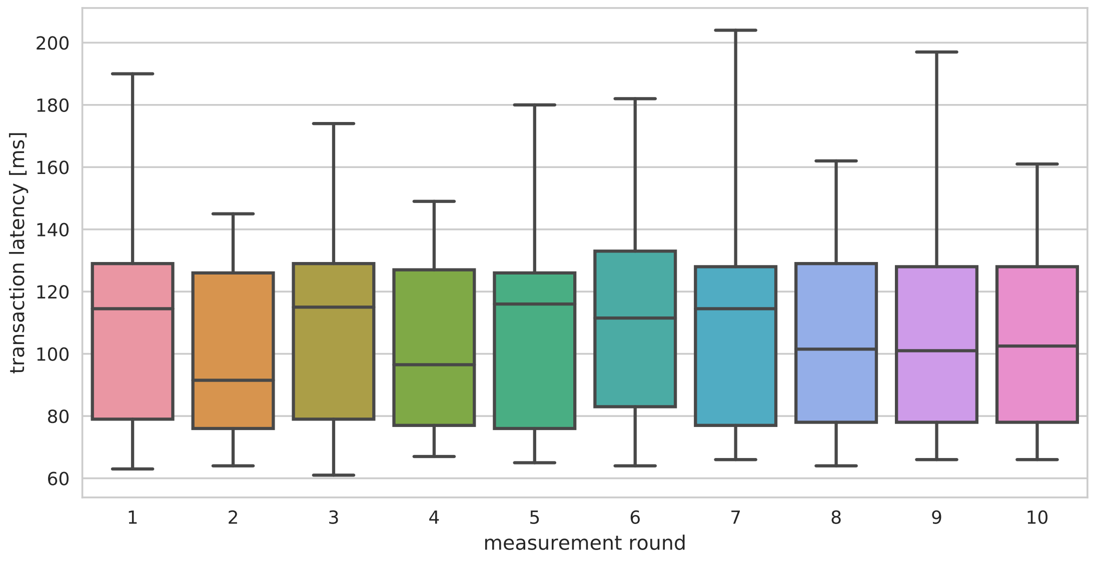

Figure 7.28: Transaction latencies with 0% network loss

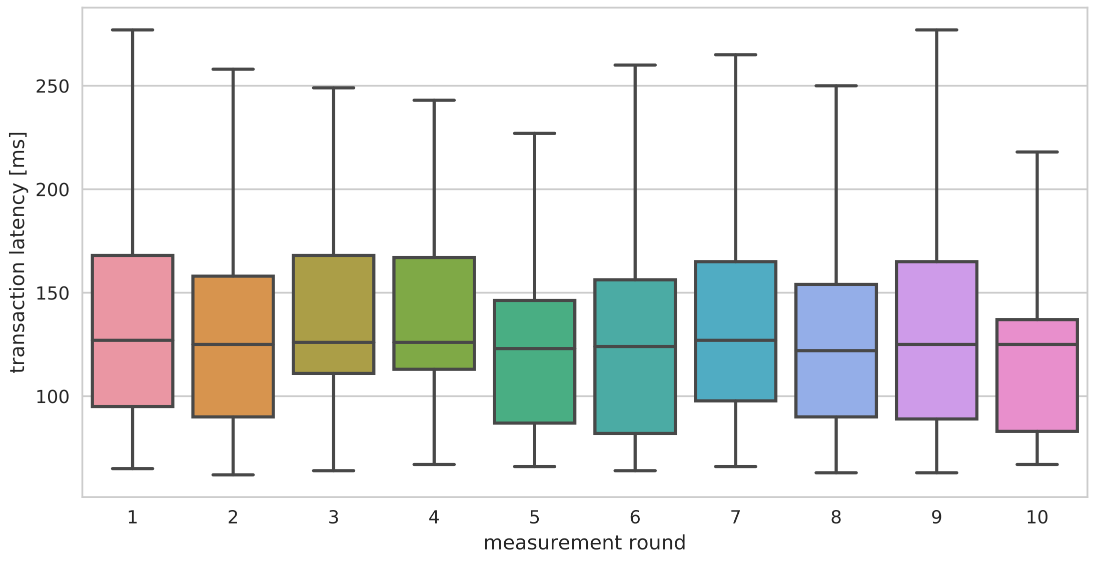

Figure 7.29: Transaction latencies with 2% network loss

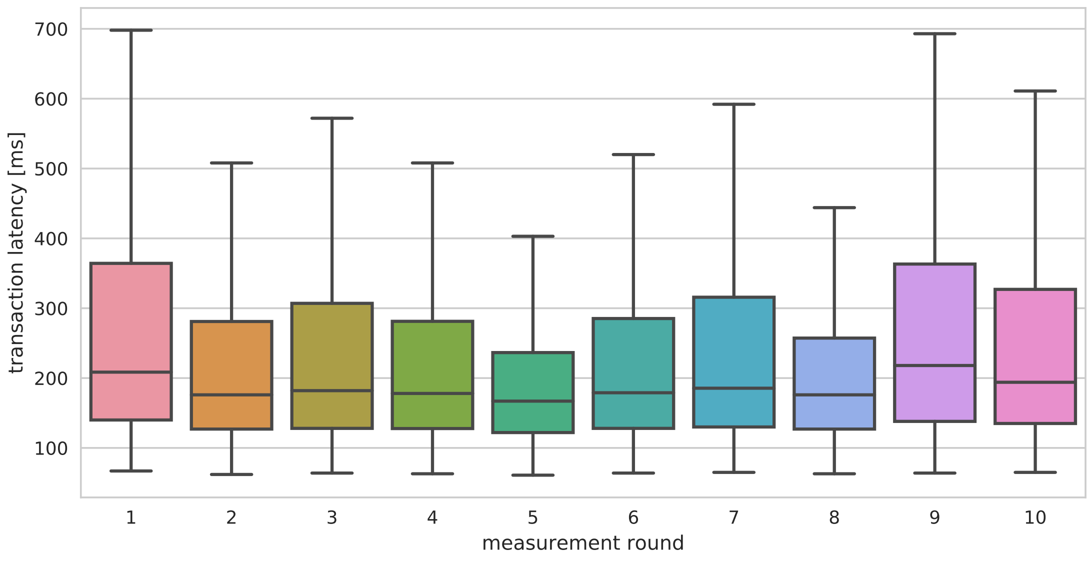

Figure 7.30: Transaction latencies with 5% network loss

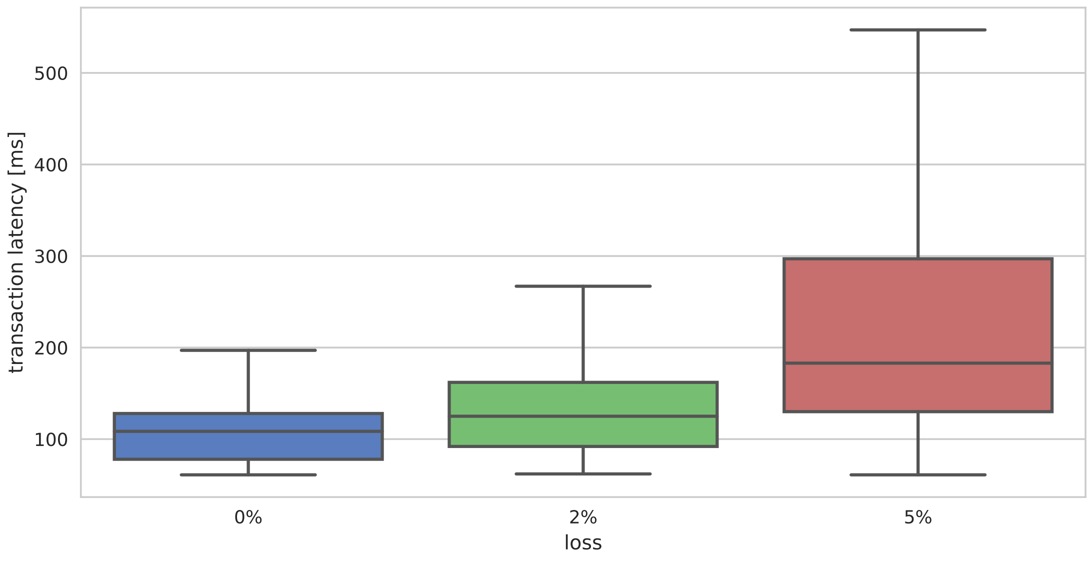

Figure 7.31: Transaction latencies with varying network loss

Chapter 8

Conclusion

In this final chapter a summary of this thesis is provided. Additionally the previously defined research questions are answered with help of the evaluation outcome. In the final section future work is proposed which might further improve the framework.

8.1 Summary

This thesis paper designed and developed a framework for private distributed ledger performance evaluation. For this purpose different layers have been defined. They range from the network layer, over the node layer and ledger layer to the application layer. For each of these layers metrics and factors have been identified, which allow to measure / influence the performance of DLT networks. Additionally workloads have been defined, which either stress individual aspects of a DLT or represent realistic use cases.

The generic implementation of the framework offers the full stack from deployment, over measurement execution, to evaluation of DLT. It allows to utilize the defined metrics and factors and supports execution of arbitrary workloads. The framework utilizes the tools Ansible and Caliper to allow easy extensibility to further DLTs. For the purpose of evaluation the DLT Hyperledger Fabric has been integrated into the framework. This allows exemplary evaluations to verify the frameworks functionality and also provides insights into Fabric's performance.

Besides an explanatory measurement, four experiments have been conducted. Their results show the impact of factors on different layers on the performance of Hyperledger Fabric. The factors transaction rate, chaincode, block size and network loss have been varied and their influence has been measured successfully and reproducibly.

8.2 Findings

This section summarizes the outcome of the evaluation and answers the remaining research questions defined in section 1.1.

What are the bottlenecks of distributed ledgers?
Section 7.2 shows that the ledger systems builds up a backlog under a continuous transaction rate of about 300 tps. The subsequent experiments illustrate how this point is moved. While a load reduction in the form of the DoNothing chaincode causes a delayed backlog build-up, putting additional load on either the node or the network results in earlier backlog accumulation. Section 7.3 has shown a slight increase in transaction latency by the DataHeavy workload, parameterized with a payload of 10 KB. The CPUHeavy workload shows an earlier and more significant increase in transaction latency. In addition to this, section 7.4 shows that a reduced block size impairs the performance immensely for high transaction rates. The reduction from 10 transactions per block to 5 transactions leads to a doubling of the mean transaction latency at a high transaction rate. In section 7.5 the impact of network loss on the performance is evaluated. The measurements show that, for a low rate of 100 tps which does not result in a backlog under normal circumstances, a network loss of 5% results in an increase in transaction latency of about 60% and even leads to dropped transactions.

Increased transaction rate, consuming workloads, misconfigured block size and high network loss have been confirmed to limit the performance of the distributed ledger, but none of them stands out as a universal bottleneck. Instead it shows that performance is dependent on different layers. Some of them can be modified to improve performance, like through adjustment of the block size to the individual requirements of the ledger application, while others might not be easily controllable like the workload or the network characteristics the distributed ledger is running on.

Is it possible to predict performance based on network and ledger parameters?
The measurements show reproducable results, which promises predictability of the performance based on the gathered results. But the number of factors that might influence the performance described in section 3.2 suggests that a model taking all of them into account would be highly complex. Instead a reduction of parameters might allow predictions for specific changes applied to the ledger network. Increased usage and thus transaction rate or also modification of the smart contracts can be predicted. The influence of a change in the smart contract's code can be represented through the elementary workloads evaluated in section 7.3. This allows to project the effect of an addition of a CPU-intensive function to a smart contract for example. For all of these predictions it has to be kept in mind, that they are based on measurements conducted in a testbed. This leads to the final research question.

What are the limits of performance evaluation inside a testbed?
First the private testbed is by default not exposed to background traffic. While it is an option to simulate background traffic, it is impossible to completely represent the real environment. Experiments in a testbed give indications on the performance impact, but applications should

be re-evaluated in the environment they are actually deployed in. Secondly the current testbed setup used during evaluation is limited in scalability. The evaluations in this thesis used a setup with only six nodes, but it is possible to increase the network size to the hardware limit in the iLab testbed. While this might not suffice for all distributed ledger use cases, it covers a large amount of applications.

8.3 Future Work

The framework developed in this thesis is built to be extended. This includes adding factors, metrics, workloads, and more: Factors could be extended by further network limitations like restricted network rate or network delay. Metrics should include information about the load on the individual nodes, to allow to make proper statements about the bottlenecks of the distributed ledger. This feature has been partially implemented in this work, but produced inconsistent results, which is why it was not included in the evaluation. Also the amount of workloads can be increased. This especially includes the simulative key value store workload described in section 3.2, where implementation has begun. To support the full feature-set of YCSB, a port of the C implementation would be required.

Additionally the number of participating nodes could be increased. Some experiments have already been run on the current testbed with multiple isles, but the results were not yet reliable. Thus, scaling has to be looked into and might require a different network setup to achieve the desired results.

Furthermore with additional results it would be interesting to evaluate to what extent it is possible to tune the distributed ledger. Especially the factors of the ledger layer could be tuned based on measurement results to adapt it to the specific environment it is running in.

Bibliography

[1] Mark Walport. *Distributed ledger technology: Beyond block chain.* 2016.

[2] Michael Mainelli and Mike Smith. "Sharing ledgers for sharing economies: an exploration of mutual distributed ledgers (aka blockchain technology)". In: *The Journal of Financial Perspectives* 3.3 Winter (2015), pp. 38–69. ISSN: 20498640. URL: `https://www.gfsi.ey.com/the-journal-x.php?pid=18{\&}id=110`.

[3] Daniel Genkin, Dimitrios Papadopoulos, and Charalampos Papamanthou. "Privacy in Decentralized Cryptocurrencies". In: *Commun. ACM* 61.6 (2018), pp. 78–88.

[4] Sunny King and Scott Nadal. "PPCoin: Peer-to-Peer Crypto-Currency with Proof-of-Stake". In: *Ppcoin.Org* (2012). ISSN: 1098-6596. DOI: `10.1017/CBO9781107415324.004`. arXiv: `1703.04057`.

[5] Elli Androulaki et al. "Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains". In: *Proceedings of the Thirteenth EuroSys Conference.* 2018, 30:1–30:15. DOI: `10.1145/3190508.3190538`. arXiv: `1801.10228`.

[6] Kostas Christidis. *A Kafka-based Ordering Service for Fabric.* last visited on 2018-07-08. URL: `https://docs.google.com/document/d/1vNMaM7XhOlu9tB_10dKnlrhy5d7b1u8lSY8a-kVjCO4/edit`.

[7] Valentin Hauner. "Trustworthy Configuration Management with Distributed Ledgers". MA thesis. 2018.

[8] Alina Quereilhac. "A generic approach to network experiment automation". PhD thesis. 2015.

[9] George F. Riley and Thomas R. Henderson. "The ns-3 network simulator". In: *Modeling and Tools for Network Simulation.* 2010. ISBN: 9783642123306. DOI: `10.1007/978-3-642-12331-3_2`. arXiv: `arXiv:1406.0440v1`.

[10] András Varga and Rudolf Hornig. "An Overview of the OMNeT++ Simulation Environment". In: *Proceedings of the First International ICST Conference on Sim-*

ulation Tools and Techniques for Communications Networks and Systems. 2008. ISBN: 978-963-9799-23-3. DOI: `10.4108/ICST.SIMUTOOLS2008.3027`.

[11] Nikhil Handigol et al. "Reproducible network experiments using container-based emulation". In: *Proceedings of the 8th international conference on Emerging networking experiments and technologies - CoNEXT '12*. 2012. ISBN: 9781450317757. DOI: `10.1145/2413176.2413206`.

[12] Brian White et al. "An integrated experimental environment for distributed systems and networks". In: *ACM SIGOPS Operating Systems Review* 36.SI (2002), p. 255. ISSN: 01635980. DOI: `10.1145/844128.844152`. URL: `http://portal.acm.org/citation.cfm?doid=844128.844152`.

[13] Franck Cappello et al. "Grid'5000: A large scale and highly reconfigurable Grid experimental testbed". In: *Proceedings - IEEE/ACM International Workshop on Grid Computing*. 2005. ISBN: 0780394933. DOI: `10.1109/GRID.2005.1542730`.

[14] Brent Chun et al. "PlanetLab: an Overlay Testbed for Broad-Coverage Services". In: *ACM SIGCOMM Computer Communication Review* (2003). ISSN: 0146-4833. DOI: `10.1145/956993.956995`.

[15] Larry Peterson et al. "Experiences Building PlanetLab". In: *Symposium A Quarterly Journal In Modern Foreign Literatures* 19 (2006), pp. 351–366. ISSN: 1012277X. URL: `http://portal.acm.org/citation.cfm?id=1298455.1298489`.

[16] P Brett et al. "Securing the PlanetLab Distributed Testbed". In: *Usenix.Org* (2004), pp. 195–201. URL: `http://www.usenix.org/events/lisa04/tech/full_papers/brett/brett_html/`.

[17] Tien Tuan Anh Dinh et al. "BLOCKBENCH: A Framework for Analyzing Private Blockchains". In: *CoRR* abs/1703.0 (2017). ISSN: 16130073. DOI: `10.1145/1235`. arXiv: `1703.04057`. URL: `http://arxiv.org/abs/1703.04057`.

[18] Yusuf Abubakar, Thankgod S Adeyi, and Ibrahim Gambo Auta. "Performance Evaluation of NoSQL Systems Using YCSB in a resource Austere Environment". In: *International Journal of Applied Information Systems (IJAIS)* 7.8 (2014), pp. 23–27.

[19] Suporn Pongnumkul, Chaiyaphum Siripanpornchana, and Suttipong Thajchayapong. "Performance Analysis of Private Blockchain Platforms in Varying Workloads". In: *2017 26th International Conference on Computer Communication and Networks (ICCCN)*. 2017, pp. 1–6. ISBN: 9781509029914. DOI: `10.1109/ICCCN.2017.8038517`.

[20] Rajitha Yasaweerasinghelage, Mark Staples, and Ingo Weber. "Predicting Latency of Blockchain-Based Systems Using Architectural Modelling and Simula-

tion". In: *2017 IEEE International Conference on Software Architecture (ICSA)*. 2017, pp. 253–256. ISBN: 9781509057290. DOI: `10.1109/ICSA.2017.22`.

[21] Ralf Reussner et al. *The Palladio Component Model*. Tech. rep. 2011. DOI: `10.1016/j.jss.2008.03.066`.

[22] Wazen M Shbair et al. "Blockchain Orchestration and Experimentation Framework: A Case Study of KYC". In: (2018). URL: `http://publications.uni.lu/bitstream/10993/35467/1/blockchain-orchestration-experimentation.pdf`.

[23] Parth Thakkar, Senthil Nathan, and Balaji Vishwanathan. "Performance Benchmarking and Optimizing Hyperledger Fabric Blockchain Platform". In: (2018). arXiv: `1805.11390`. URL: `http://arxiv.org/abs/1805.11390`.

[24] Marc-Oliver Pahl. "The iLab Concept : Making Teaching Better, at Scale". In: *IEEE Communications Magazine* November (2017), pp. 2–9.