universität wien

MASTERARBEIT / MASTER'S THESIS

Titel der Masterarbeit / Title of the Master's Thesis

„Architectural Issues regarding Blockchain Systems: A Framework for Object Relational Mapping Support for Smart Contracts“

verfasst von / submitted by

Martin Pfitscher, BSc

angestrebter akademischer Grad / in partial fulfilment of the requirements for the degree of

Master of Science (MSc)

Wien, 2021 / Vienna, 2021

Studienkennzahl lt. Studienblatt / degree programme code as it appears on the student record sheet:	UA 066 921
Studienrichtung lt. Studienblatt / degree programme as it appears on the student record sheet:	Masterstudium Informatik
Betreut von / Supervisor:	Univ.-Prof. Dipl.-Ing. Dr. Wolfgang Klas
Mitbetreut von / Co-Supervisor:	Dr. techn. Belal Abu Naim

Acknowledgements

I would like to thank my thesis supervisor Univ.-Prof. Dipl.-Ing. Dr. Wolfgang Klas of the University of Vienna. His expertise and inputs were very helpful when it came to writing the thesis. Additionally, I would like to thank my co-supervisor Dr. techn. Belal Abu Naim of the University of Vienna for supporting me from my first blockchain-related project in the "Praktikum 1" until the end of my master thesis. This thesis would not have been possible without his input, expertise, and motivation. He always found time for discussing the thesis when I had questions.

Abstract

Recent years have shown many developments in the direction of immutable and distributed storage. One of them was the development of blockchain databases. These systems combine blockchain and database features while trying to remain competitive. Some of these systems are similar to traditional databases by supporting Structured Query Language (SQL) like language or using traditional databases as storage. Nevertheless, there is no general interface for blockchain databases. A missing general interface in blockchain databases leads to a steep learning curve and requires custom solutions with high maintenance costs. While typical databases have standardized properties such as SQL they greatly benefit from Object Relational Mapping (ORM) to reduce development and maintenance costs. This thesis investigates if an extendible ORM solution based on code generation is feasible for blockchain databases. Additionally, we discuss the advantages and disadvantages of using a blockchain-based system as general-purpose storage. This thesis investigated multiple blockchain database storage options, as the properties highly depend on the underlying storage system and the communication between the components (i.e., consensus). As proof of concept, this thesis implements two blockchain database showcases for our Smart Contract Object Mapping (SmartCOM) framework. SmartCOM proposes a general-purpose storage architecture supporting Create Read Update Delete (CRUD) and trace capabilities for both showcases. The two showcases are then compared against each other and vaguely to similar state-of-the-art frameworks. As a result, these showcases are interchangeable. Additionally, SmartCOM is extendible for further blockchain storage options. With SmartCOM, users can already store data on two different blockchain databases without blockchain programming knowledge or writing platform-specific code.

Kurzfassung

Die vergangenen Jahre zeigten eine Entwicklung zu unveränderbaren und verteilten Speichersystemen. Eine davon waren Blockchain-Datenbanken. Diese Systeme versuchen Eigenschaften traditioneller Datenbanken mit denen von Blockchains zu vereinigen. Einige dieser Systeme nutzen Structured Query Language (SQL) ähnliche Abfragesprachen, während andere gänzlich auf traditionelle Datenbankspeicher setzen. Allerdings gibt es für diese Systeme noch keine allgemeine Schnittstelle. Bei traditionellen Datenbanken gibt es hingegen das Prinzip des Object Relational Mapping (ORM), um die Komplexität zu verbergen und das Wechseln zwischen verschiedenen Systemen zu vereinfachen. ORM bildet Objekte einer Programmiersprache auf Datenbanken ab, sodass ein Speichern und Manipulieren solcher Abbildungen möglich wird. Diese Thesis untersucht die Möglichkeit, das ORM-Konzept auf Blockchain-Datenbanken anzuwenden. Zusätzlich werden Vor- und Nachteile der Speicherung von generischen Daten auf Blockchainsysteme diskutiert. In dieser Arbeit werden mehrere Optionen für Blockchain-Datenbanken besprochen, um zu zeigen, welche Auswirkungen die ausgewählten Speichersysteme sowie andere Komponenten (insbesondere Konsensus-Mechanismen) auf die Eigenschaften einer Blockchain-Datenbank haben. Als Beispiele für unser Framework Smart Contract Object Mapping (SmartCOM) wurden zwei Blockchain-Datenbank Systeme für SmartCOM konzipiert. Mithilfe dieser kann der Nutzer zwischen zwei verschiedenen Blockchain-Datenbanken wählen, ohne plattform-spezifischen Quelltext zu schreiben. SmartCOM unterstützt die Funktionen, Elemente zu speichern, ändern, löschen und aus dem Speicher auszulesen. Des Weiteren wurde eine Funktion, die es erlaubt, Änderungen bei gespeicherten Elementen nachzuverfolgen, bei beiden Blockchain-Datenbank Prototypen implementiert. Die beiden Prototypen werden gegenübergestellt und grob mit ähnlichen Blockchain-Datenbanksystemen verglichen. Dadurch, dass beide Prototypen dieselben Methoden unterstützen, sind sie untereinander austauschbar und es wird gezeigt, dass weitere prototypische Implementationen unterstützt werden können.

Contents

List of Tables

List of Figures

Listings

1. Introduction

The recent developments in blockchain technologies have opened new data storage possibilities. Many state-of-the-art blockchain systems support smart contracts, which allow the storage of more complex data. Furthermore, smart contracts allow programmers the integration of business logic into the blockchain. Therefore, it is possible to build sophisticated applications. An often-used example is a supply chain network that shares data over multiple (possibly untrusted) parties, and changes need to be traceable. Typically a blockchain-based system needs to be created from scratch and optimized for a single application. While this may result in an efficient and performant application, it often involves custom solutions and non-reusable code. Therefore, blockchain databases are developed to store data in a more generic and database-like manner. Blockchain databases are categorized into systems with blockchain storage (e.g., SEBDB [ZZJ+19]) and systems that imitate blockchain storage (e.g., BigChainDB [Gmb18]). The main difference is that the systems that imitate blockchain storage use structured query language (SQL) or no-SQL storage and try to reproduce the blockchain properties with other measures (e.g., by adding a consensus protocol [Gmb18]). These blockchain-specific properties include decentralization, fault tolerance, immutability, and redundancy. The systems with blockchain storage are based on a blockchain and therefore have the blockchain properties mentioned above. Nevertheless, they need to implement at least some of the database properties needed for a blockchain database. These properties include indexing and querying support, the possibility of data confidentiality, and high write/read performance [CCK+18, Gmb18].

As some of the properties of a blockchain clash with database properties, there will not be an optimal solution for every use case. One of the drawbacks of using blockchain-based storage is that no general query language or storage architecture exists. Typically, every smart contract-based blockchain system has its programming language and constructs. Additionally, each blockchain system has its application programming interface (API). Thus, switching between different blockchain systems requires rewriting large parts of an existing project. Object Relational Mapping (ORM) solved a very similar problem, allowing conversion between incompatible type systems using object-oriented languages (e.g., Hibernate ORM). By using ORM, a user can switch with little effort between different SQL/no-SQL databases. This thesis aims to create a framework called Smart Contract Object Mapping (SmartCOM) which is a similar solution for blockchain-based storage. Advantages of using such an approach include overcoming vendor-specific differences and lowering developing time and costs. Another advantage is that if SmartCOM supports a newer version of a blockchain API or smart contract language, the user project will benefit from the more recent version.

Assume that the government health organization uses a traditional relational database

ORM system to manage the vaccination certificates. To allow private doctors to vaccinate, they share access to their system, especially their patient's medical data. Private doctors then register new vaccination. Imagine one of the private doctors acts maliciously and adds vaccinations that never happened. The government may have vaccination dose identifiers to prevent this. A doctor can give a single dose only to a single person with traceability of the origin of the vaccine and the location and time of usage. This becomes a problem if a doctor decides to remove (or delete) these vaccination entries from the system. After removing them, the vaccine is still in the system, but it appears unused. The doctor could then add them to his not vaccinated clients without actually vaccinating them. Typically, a specific person has access to this information, or a log file exists. But if the person with access to this data plays along with the malicious doctor and deletes or alters this data, it would be catastrophic. As a result, the unvaccinated patients could be in close contact with other people disregarding pandemic rules. Another problem could be that people who were in reality vaccinated may lose their digital certificates (as the malicious patients got their certificate).

By changing to blockchain-based solutions, the blockchain guarantees data immutability. Naturally, it does not ensure that the inserted data is correct. But in our identifier example, it would be visible that a dose exists twice, and the malicious doctor is traceable with no way of deletion. Another problem to address is the transition from the existing relational database to a blockchain system. A replay function is needed to facilitate this, adding all the relational database data with the timestamps (and in the same order where required). As adding wrong timestamps should not happen in a blockchain environment, a function is necessary to disable this feature. In principle, this requires only a flag called data migration which starts with a value of true. After migrating, it is vital to have a single function that can only alter the flag to false. After the migration, all timestamps need to be generated by the system. The blockchain also guarantees that these functions are immutable (until redeploying them).

An essential contribution of this thesis is to show that a blockchain-based ORM solution is possible and feasible. Other necessary parts include showing implementation options, demonstrating problems, and indicating future work. While the SmartCOM implementation in this thesis provides support for two blockchain systems out of the box, the proposed framework is extendible with other blockchain systems and customizable if needed.

The following chapters should demonstrate the capabilities and challenges of SmartCOM. Chapter 2 discusses the background of blockchain technology, the abstract principles behind ORM, and the possibilities for user interaction with an ORM system. This background information is the foundation to evaluate and understand the state-of-the-art. Chapter 3 discusses the scientific and commercial approaches to blockchain systems that work as a generic database. Additionally, we will discuss selected code-generation/model-mapping approaches for blockchains. While there is no direct connection between these two fields, SmartCOM contains parts of both. To understand the differences of each field, we will compare them beforehand among their peers and then discuss their impact or connection to the proposed solution. Chapter 4 then discusses the framework specification

by discussing the SmartCOM architecture and architectural concepts shared among all blockchain implementations supported by SmartCOM. Chapter 5 then discusses the two blockchain implementation showcases for SmartCOM. As a blockchain is not automatically general-purpose storage for user-defined models, we present a smart contract architecture for the two showcases. These showcases are then evaluated for their performance and compared to other systems. Finally, in Chapter 6, we conclude the thesis with our findings, limitations, and outlook on further research questions and possibilities to extend SmartCOM.

2. Background

In order to understand the design decisions, limitations, state-of-the-art, and properties of SmartCOM we first need to understand the technological background. Therefore, we will discuss important properties of blockchains, databases, and ORM frameworks. Most of these properties can be seen as tools, that depending on the requirements, can be used to build a framework. Although we decided on properties when creating SmartCom, it may not be the optimal configuration for each Use case. A variation selecting different properties may lead to better results in certain use cases (e.g., different mapping approaches). The next sections describe blockchain technology, relational databases, ORM approaches, data access options as well as metadata mapping and retrieval approaches.

2.1. Blockchain

Blockchain technology is widely known for its use in implementing cryptocurrencies often exchangeable for traditional currencies. Besides providing cryptocurrencies, blockchains are also capable of storing data in a fault-tolerant and redundant manner. A blockchain consists out of a list of blocks that contain transactions. To make the blockchain immutable, each block stores, among other information, the hash of the previous block. Some blockchains like Ethereum or Hyperledger allow smart contracts. Smart contracts are computer programs that can enforce rules (e.g., how to transfer a cryptocurrency) and change an object's state on the blockchain. All modifications occurring during a smart contract are recorded. Therefore, old object/smart-contract states are never lost and always traceable.

2.1.1. General Blockchain Properties

This subsection describes general blockchain properties and how it compares to a centralized database.

Decentralization A blockchain operates without any single governing authority. It works by having a group of nodes that make decisions and run the blockchain. A database typically needs a central trusted party.

Robustness/fault tolerance In a blockchain system, data gets distributed among nodes. Additionally, blockchain systems typically enforce Byzantine Fault Tolerance (BFT). BFT means that less than one-third of the nodes can fail in any way, and the system will still be able to agree on how to proceed. Centralized databases store data in a centralized manner, not restricted to a single physical site (e.g., a single Server).

Nevertheless, storing all data in a centralized way introduces a single point of failure (e.g., if an interface for storing is no longer reachable). As a result of a centralized database failing, the database is no longer available for applications.

Immutability A blockchain system writes data in an append-only manner into blocks containing transactions. Once a transaction gets stored, it cannot be deleted or modified. Modifications of the state can only occur due to new transactions (e.g., using smart contracts). These changes will be recorded and are therefore traceable. In a typical database, data can permanently be changed or deleted.

Owner-controlled assets In a blockchain, only asset owners can transfer assets. An owner is a holder of a set of private keys. For example, it is not possible to transfer Bitcoin without a private key. Databases have support for user authentication, role, and group management. The main difference here is that a user does not need to sign a transaction. Therefore, a transaction's sender is not stored in a persistent (and immutable) manner by default.

Redundancy In a blockchain system, each participating node has a copy of the latest data. However, in a centralized database, only the central party has a copy of the latest data. Furthermore, as not all applications need all data on-site, a data storage central location may also be beneficial due to storage (i.e., number of hard drives) and synchronization requirements.

Digital signatures Blockchain uses cryptographic measures by default. For example, transferring a cryptocurrency (e.g., Bitcoin) to another cryptocurrency wallet requires the private key of the owner. Thus, the transaction is signed by the owner and thereby secured. On the other hand, a database may use more traditional access control. For example, in a database, a representation of a coin may not require a digital signature of the owner to be spent.

2.1.2. Consensus Mechanisms

The survey of Xiao et al. [XZLH20] mentioned five components of a blockchain consensus protocol. The first of them is the block proposal which describes the block generation process. Secondly, we need information propagation as blocks and transactions get sent over the network. Thirdly, blocks need a validation process to guarantee the validity of the blocks and the enclosed transactions. Fourthly, based on block validation as the nodes in the network need to agree on accepting valid blocks. Finally, a mechanism to reward participants (especially miners) and to create tokens (e.g., as a block reward) is needed. In this subsection, we will discuss three consensus mechanisms.

Proof of Work

In Proof of Work (PoW) an accountant is selected by mining. In PoW blocks are added directly after their verification. Transactions are confirmed via the longest chain principle,

which means that the longest chain of blocks which contain transactions (i.e., chain with most work) gets selected as the main chain. PoW is, for example, used in the Bitcoin network [XF21].

Proof of Stake

In Proof of Stake (PoS) a node is selected with a higher probability as an accountant if it has more stake. The stake is typically measured by tokens held or a combination of the amount and age of the tokens held. Similar to PoW, blocks are added directly after verification, and for transaction confirmation, the longest chain principle is used. While Ethereum is most known for its usage of PoW, it is also shifting to PoS [XF21].

PBFT

In Practical Byzantine Fault Tolerance (PBFT) an accountant is selected by having a poll between all nodes. Block addition is done after verification and a three-phase commit (pre-prepare, prepare, and commit). Blocks and their transactions will be confirmed when they are added to the blockchain [XF21].

2.1.3. Blockchain Types

There exist three main blockchains types. These types affect their consensus determination, read permission, immutability, efficiency, centralization, and consensus process. [Zhe18]

Public Blockchain

A public blockchain is open to the public in terms of reading and sending transactions. Everyone can take part in the consensus mechanism. Instead of being centralized, cryptographic verification (e.g., PoW) and economic interests (i.e., mining cryptocurrency) ensure trust (and security). These instruments provide that financial resources limit the degree to which someone can influence the system [Blo15].

Consortium Blockchain

A consortium blockchain limits access to their consensus protocol to a set of selected nodes. It can be imagined by thinking of a consortium where every hospital in a country runs a node. As the nodes are selected, it is a partially centralized system. The read access in a consortium can be public or restricted to specific participants [Blo15].

Private Blockchain

One centralized organization has the write permission in a private blockchain, while reading may be public or restricted. This type may be interesting for single companies for auditing or data management [Blo15].

2.1.4. Overview

Zheng et al. [Zhe18] provided a comparison between the different blockchain types. Table 2.1 shows their differences. While a public blockchain is nearly impossible to tamper with, it is also relatively inefficient. If efficiency is the main goal or data needs to be restricted, then a consortium or a private solution would be beneficial. A drawback of this centralization is that the centralized party may accept only beneficial blocks. Therefore it is important to keep track of the participating parties and only allow trusted ones. Otherwise, it may lead to immutability problems. In a public blockchain, this is of minor concern as there is a monetary incentive. Nevertheless, a public blockchain needs multiple parties and should never have a party more influential (controlling more of the blockchain) than all other participants combined.

Property	**Public**	**Consortium**	**Private**
Consensus determination	All miners	Selected set of nodes	One organisation
Read permission	Public	Public or restricted	Public or restricted
Immutability	Nearly impossible to tamper	Could be tampered	Could be tampered
Efficiency	Low	High	High
Centralised	No	Partial	Yes
Consensus process	Permissionless	Permissioned	Permissioned

Table 2.1.: Blockchain type comparison taken from [Zhe18]

2.2. Relational Database

In this section, we will discuss three relational database properties that would also be desirable for blockchain technology.

Confidentiality of Data

In a database, access policies for specific data can restrict access to authorized persons. Similarly, there exist some permissioned blockchain systems that support the notion of confidentiality [NGS+19]. Note that confidentiality is, to some extend, clashing with the concept of traceability by all participants.

Performance

In blockchain systems, it may take a long time to confirm transactions. In Bitcoin, confirmation means that a transaction is added to a block and included in the public ledger. While Bitcoin took an average time of around ten minutes [bit21a], Ethereum

took around 13 seconds to mine a block in February 2021 [bit21b]. These blocks are build of a set of unconfirmed transactions. After other miners confirm the block, the block becomes an immutable part of the blockchain as subsequent blocks confirm the existence of the previous blocks. Although Ethereum and Bitcoin blocks are not directly comparable (i.e., block size, number of contained transactions in block), it indicates a public network's performance. On the other hand, a database has a high transaction rate, high capacity, and low latency, which yields an immediate execution/update [CCK+18].

Indexing and Querying

Traditional blockchain systems support only very restricted indexing and querying (e.g., retrieving blocks by block number). Databases, on the other hand, can support complex queries and support various indexing techniques [CCK+18].

2.3. Object Relational Mapping Background

Object Relational Mapping (ORM) describes the process of linking Object Oriented Programming Language (OOPL) to relational databases. This is accomplished by establishing a connection between classes of the programming languages and tables in the database [LHR16].

2.3.1. ORM Concerns

First of all, relational databases and OOPL have different paradigms. Relational databases define a set of relations, while OOPL define a network of related objects. Secondly, incompatibility of data structures between relational databases and OOPL is a concern. Additionally, representations of the same object in two different languages, the host OOPL and the database query language, need to be maintained. At last, the storage and retrieval of these objects need to be handled [IBNW09].

2.3.2. Mapping Considerations

There are different ways to map objects from a OOPL to relational database tables. At first, we look at a single class without relationships. Then, after laying the basis, we extend the single class mapping to include relationships.

Single Classes

For single tables, there exist mainly four mapping options [Ors06, Fow02]. The first approach is called single table inheritance and consists of mapping a class hierarchy into a single table. A single table includes all attributes in a class hierarchy, including parent classes and other concrete classes within the same hierarchy. An example would be a student and teacher class which are both inheriting the person class. The result would be one table with all attributes defined in the three classes, as mentioned earlier. The

second approach is called class table inheritance and consists of mapping every class into its own table. In the student, teacher, and person class example, we would have three distinct tables connected via foreign keys. The third approach is called concrete table inheritance, which works by mapping every concrete class into its own table. In our three-class example, we would have two tables. These are representing the student class and the teacher class, both containing the attributes of the person class. The fourth approach stores any objects in a generic table structure to handle any kind of generic classes. Imagine the table storing attributes as text strings with their respective attribute types. Retrieving data would require a large amount of typecasting.

Relationships Between Classes

As classes in an object-oriented programming language can have relations to other classes, we need a strategy to map those relations to the concept of relations of a relational database. In the case of many-to-one and one-to-one relationships, a single-valued field stores the relationships' foreign key (e.g., an integer identifier). One-to-Many relationships can be handled with a collection field storing the different foreign keys (e.g., a list of identifiers). Finally, many-to-many connections use an association table mapping. An association table is an additional table that stores the foreign keys of both sides of the relationship [Fow02].

2.4. Data Access

2.4.1. Data Access Object

The Data Access Object (DAO) pattern decouples the object-oriented language from the data source. As the user/programmer directly interacts with the API of the DAO and not the underlying data source, it is easier to change the underlying database. The DAO hides all implementation details from the business object. A business object never interacts directly with the data source. The only way to manipulate or read the stored objects/entities in the data source is to use the data access object. A representation of the data in the data source called transfer object allows DAO pattern to convert the OOPL classes to the data source representation [Ber05]. Instead of directly changing the data source objects/entities, a representation of the data stored, called transfer object, must be changed and passed to the data access object. Also, for reading, only transfer objects get returned to the business object. A DAO is stateless. Stateless means it does not cache the results or parameters of a query (after the execution) [AMC01].

2.4.2. Repository

A repository is a collection-like interface with more elaborate querying capabilities. Clients can request objects by matching some criteria (e.g., an identifier) from a repository. A repository seems like a collection capable of adding or removing instances, while in reality, they are inserted or deleted from the database [Eva04, Fow02, AB20]. A repository and

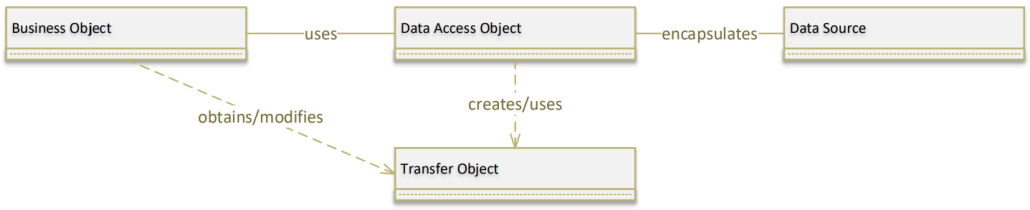

Figure 2.1.: Representation of DAO pattern from [Ber05] (altered color scheme)

a DAO's main difference is that a DAO is at a lower abstraction level than a repository. A DAO is an abstraction of data persistence, while a repository abstracts a collection of objects.

2.5. Metadata Mapping

Metadata mapping describes how a running program uses metadata information about how object fields map to columns in a data source [Fow02]. The following approaches vary in complexity and execution time (i.e., if executed before or when running a program). However, one of the benefits of these approaches is that it omits repetitive code.

2.5.1. Code Generation

Code generation means that during the build process, source code gets generated from metadata information. This source code might look handwritten, but changes by hand should never happen. A code generation approach's drawback is that it is less dynamic since it requires recompiling and redeploying [Fow02].

2.5.2. Reflective Programming

Reflective programming describes a program's ability to manipulate and read the program's own state during runtime. The capabilities include intercession and introspection. Intercession represents the aspect of modifying the own program state. In contrast, introspection describes the element of reading the own program state. An example of introspection is if an object has a getter (e.g., getName) for an attribute in an object-oriented programming language. While reflection has some speed issues, it does not matter as much compared to the slow remote calls to the data source. Furthermore, compared to code generation, which needs to gather metadata information at build time, reflection will typically gather metadata information at run time [GWB91, Fow02].

2.6. Metadata Retrieval

As metadata mapping requires metadata information, a way to gather that information needs to be selected.

2.6.1. Separate File

Metadata can be defined using a separate file. One of the most used file formats is Extensible Markup Language (XML). During metadata mapping, these hierarchically structured files get parsed and used [Fow02].

2.6.2. Source Code

Source code can define metadata for metadata mapping. An example of such metadata would be Java Annotations, a form of syntactic metadata added to the source code (e.g., at class level) [Fow02].

2.6.3. Data Storage

Another option would be to store the metadata directly in the data storage (i.e., database). As the metadata describes the mapping of classes in a ORM to a data source, it would not require additional elements [Fow02].

3. State of the Art

In the previous chapter, we discussed the different properties of blockchain, database, data access, and Object Relational Mapping (ORM) technology. This chapter will use these properties to describe how the current approaches combine ORM and code generation with blockchain technology. Additionally, we describe the recent development of combining blockchain and database properties called blockchain databases. One reason to discuss blockchain databases is that SmartCOM could use some of these solutions directly. Another reason is that there are many exciting concepts within the approaches. An example of such an approach is to support off-chain data while guaranteeing that critical data is immutable by linking it to on-chain data. Finally, we look into both state-of-the-art academic and commercial state-of-the-art approaches to give a broader view of this novel research field.

3.1. Blockchain Database

A blockchain database is a concept of combining the properties of traditional databases and blockchains. Blockchain databases can be categorized into systems imitating blockchain storage and systems using blockchain storage (also known as blockchain-based storage [com20]). In this section, we will analyze these two blockchain database approaches as well as their implementations.

3.1.1. When to Use Blockchain-Based Storage

Chowdhury et al. [CCK+18] developed a diagram to help users decide when to use a blockchain, whether it should be private or public, and whether to store data on or off-chain. Their conclusion was to use a blockchain only when having multiple parties with a trust deficit, with no trusted third party, where transactions records should be maintained immutable, and scalability is no critical requirement. After meeting all these conditions, it depends on whether it is essential to verify data publicly. If this is true, the authors suggest a public blockchain, otherwise a private one. Finally, it depends if data durability is vital. If this is also the case, then store the data on-chain otherwise off-chain.

When looking at all the points where a database would be better than blockchain-based storage, it raises the question if it is possible to combine existing solutions to use the advantages of both systems. In the following section, we discuss some of the proposed approaches.

3.1.2. Blockchain Systems Imitating Blockchain Storage

Systems that imitate blockchain storage use structured query language (SQL) or no-SQL storage and try to reproduce the blockchain properties with other measures. These approaches use a similar consensus mechanism as a blockchain to guarantee the databases' immutability and integrity of data held at each node. The databases often store the data in blocks containing transactions to facilitate the consensus network conversion.

Order-Then-Execute Blockchain Databases

This approach uses a consensus mechanism to order transactions within a block. After finishing ordering the transactions, nodes executed them concurrently [NGS+19].

Example

BigchainDB is an open-source blockchain database [Gmb20b]. It uses MongoDB, a distributed NoSQL database for data storage. To achieve consensus, it uses Tendermint, a BFT consensus engine. According to [Gmb18] the intention is that a set of organizations controls the network. Therefore, it is either private or a consortium-based solution.

Figure 3.1 depicts a BigchainDB network with four nodes. Every node has its local MongoDB and Tendermint instance. BigchainDB provides API to post transactions and get transactions, transaction outputs, assets, and blocks [Gmb20a]. Once a transaction gets sent to a Tendermint node, it gets validated. Upon passing the check, it gets included in the transaction storage of the node. Additionally, it will be broadcasted to other peers and eventually included in a block [Ten20]. Tendermint proposes these new blocks to nodes. The nodes then agree on blocks in a BFT manner. Each BigchainDB instance keeps track of the block which is under construction. New transactions sent by Tendermint to BigchainDB instances are checked for validity and not directly written to the database. Only after a commit message does a BigchainDB instance write to its database [Gmb18].

Execute-Order-In-Parallel Blockchain Databases

The Execute-order-in-parallel principle allows nodes to execute transactions in parallel without prior knowledge of their ordering. The ordering gets determined in parallel by an ordering service [NGS+19].

Example

IBM presented in their paper *blockchain meets database* [NGS+19] a blockchain database approach based on PostgreSQL storage. A modified version of Serializable Snapshot Isolation (SSI) guarantees consistency and order determined by consensus across replicas. Like the previously mentioned BigchainDB, the IBM project targets a network consisting of known but untrusted organizations. Each Organization is operating its database nodes. Besides simple operations, the system also supports smart-contract-like PL/SQL procedures. Snapshot Isolation (SI) reads from a snapshot of the databases at the start

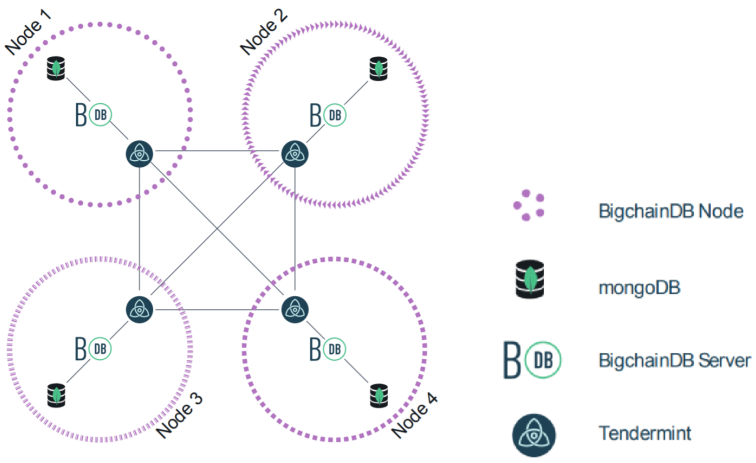

Figure 3.1.: Communication in a four-node BigchainDB 2.0 network (taken from [Gmb18] with altered legend size).

of the transaction. The problem is that SI alone does not guarantee consistency (due to anomalies). By detecting anomalies and resolving them, SSI ensures serializability. Execute-Order-In-Parallel requires transactions to include the block height (the number of preceding blocks to the block, including the transaction). The system then executes a transaction upon reaching the given block height. In parallel, the ordering service orders the transactions. After the transactions are complete, they either get committed or aborted. During this phase, conflicts between depending transactions still need to be resolved. An example of such a dependency is a write-write dependency within the same block (two or more transactions modifying the same object). Consequently, the system commits the transaction ordered as the first. Finally, the checkpointing phase computes the hash of the *write set*. This *write set* contains all changes in the database due to the new block. This hash is the proof of execution and the commit [NGS+19].

3.1.3. Blockchain-Based Databases

Blockchain-based databases store the data in a blockchain and implement at least some of the database properties. Often they are implemented as a hybrid system using both local traditional databases and a distributed blockchain.

Techniques

Blockchain Anchoring Blockchain anchoring allows the linking of two layers of blockchains. The second layer blockchain is responsible for creating transactions that record the first layer blockchain's hash in certain intervals. By doing so, it validates the integrity of the first layer blockchain [com20].

Example Gaetani et al. [com20] have proposed a blockchain anchoring technique for a cloud federation. A cloud federation allows the connection of different cloud computing environments. Their approach for blockchain anchoring consisted out of two layers. The first layer is a database interface with a database layer and a rotation-based blockchain. Rotation-based means that each blockchain miner will be the leader for a given amount of time. The leader will then sign and broadcast new operations. These operations then need to be accepted by the other miners. Every miner will then update their local ledger as well as their database replica. The second layer of the blockchain anchoring is a PoW blockchain, which is responsible for storing witness transactions (hashes) of the data in the first layer blockchain (rotation-based blockchain) in an immutable way.

Concrete real-life pilot projects are the hashing of cadastral data in Georgia and the Ukraine [SP19, KP19].

Blockchain Integrated Databases

Another possibility to create a blockchain-based database is to integrate database capabilities in the form of smart contracts. The integration can go from storing database-like objects to implementing a query system based on smart contracts. The main difference to blockchain anchoring is that we have a single blockchain storing database objects and not only their hash.

Example Helmer et al. [HRIP18] have proposed a system called EthernityDB. Their system integrated a database system into the Ethereum blockchain. Connected smart contracts represent the different database components. The system gets inspiration from the Not Only Structured Query Language (NoSQL) database MongoDB architecture, therefore, it uses document storage instead of a table structure (as in typical SQL databases). Figure 3.2 shows the smart contract architecture used to query and store data. It consists out of the following components:

Driver smart contract The driver contract is the query engine of EthernityDB. It is responsible for validating insertions, parsing queries, accessing the data stored in the document contracts, and producing query results.

Database smart contract The database contract is the entry point for all operations on a database. It contains some configuration parameters for the database and indexes collections managed by the database instance.

Collection smart contract Similar to MongoDB, EthernityDB stores multiple documents in a collection that belongs to a database. Using a single contract for manuscripts and collections lowers execution costs in terms of gas (costs for the execution on the Ethereum blockchain). Documents get represented in BSON (binary JSON) format. Each such document is split into chunks of 32 bytes to use the smart contract storage more efficiently.

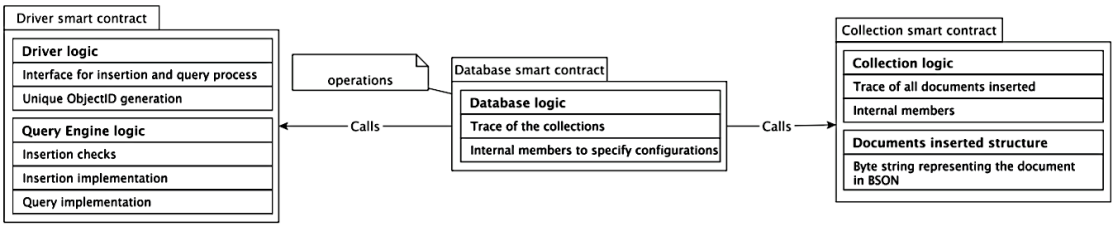

Figure 3.2.: EthernityDB architecture overview taken from [HRIP18]

On- and Off-Chain Anchoring The concept of on- and off-chain anchoring is similar to blockchain anchoring (which stores hashes of off-chain data). The difference is that the on- and off-chain anchoring stores parts of the data object unhashed and direct (e.g., a set of columns of a table) into the blockchain system. The sharing needs between the parties determine the on-chain data. This concept leaves private or unnecessary information off-chain. As no off-chain data or a hash is stored on-chain, it will not guarantee its integrity or immutability.

Example Zhu et al. [ZZJ+19] proposed a Semantics Empowered BlockChain DataBase (SEBDB). The main goals of SEBDB include adding relational data semantics to a blockchain platform and improving the execution efficiency by adding indices. It considers both on- and off-chain data. SEBDB stores and orders transactions according to their timestamp in blocks. The system introduces a table schema for all transactions of the same type. Each of these transactions has a set of columns. Some automatically added attributes such as a transaction ID, signature, sender, and transaction type exist besides the user-defined attributes. The tables' entries get handled via a SQL-like language that supports create, insert, select, join and trace (to trace events over time) statements.

Figure 3.3 shows an example for a donation application using SEBDB. Donor, project, organization, and donee id are visible for all participants of the blockchain network. By that, SEBDB can transfer a digital representation of money between the organizations. More private information such as the e-mail address is only stored by the organization that is responsible for it.

3.1.4. Blockchain Performance Improvements

In this section, we will discuss a very promising blockchain architecture. With this architecture, a blockchain database based on smart contracts would become more feasible. This approach is no blockchain database as it does not support SQL-like querying, but can lay the basis for it.

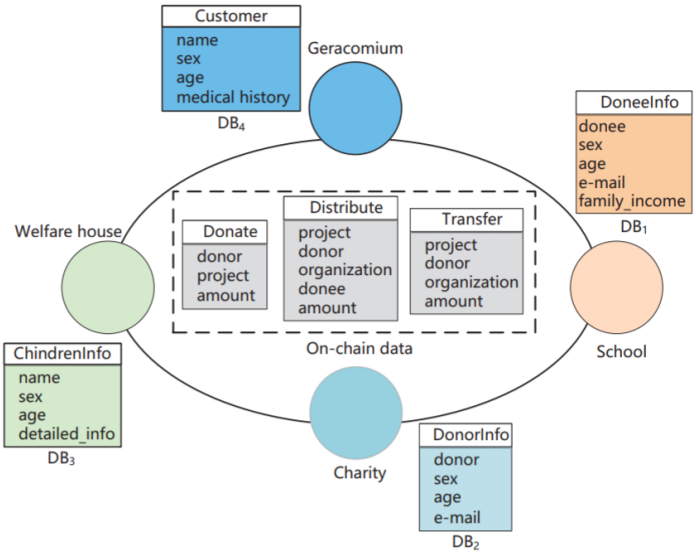

Figure 3.3.: Example of usage of on and off-chain storage taken from [ZZJ+19]

Hyperledger Fabric

Hyperledger Fabric is an open-source system for developing and operating permissioned blockchains. It does support modular consensus protocols. It runs general purpose programming languages without a build-in cryptocurrency. It also supports identity management [ABB+18].

To understand the performance difference between traditional blockchains and Hyperledger Fabric, we need to understand the order-execute model used by most other blockchain systems. Firstly, the blockchain network orders transactions with consensus or atomic broadcast. Secondly, transactions are executed in the same order and sequentially on all peers. At last, all peers persist in the state. If we take Ethereum proof of work blockchain, we see in principle the same order-execute model. In Ethereum, every participating peer assembles a block with valid transactions. The peer then tries to solve a proof of work puzzle. If the peer solves the puzzle, it distributes the block to the network via a gossip protocol. At last, every peer validates the solution to the puzzle and all transactions within the block. As every peer runs the transactions in the same order sequentially, it fulfils all order-execute architecture properties. The order-execute architecture is well understood and, due to its simplicity, widely used. But there are some issues, especially concerning performance. First of all, transactions get executed in sequential order. This limits the throughput as it can become a bottleneck for smart contract execution. Another problem is if a blockchain network (as in Hyperledger) lacks the principle of a cryptocurrency, an adversary could deploy a smart contract with an infinite loop to achieve a denial-of-service (DoS) attack. In blockchain systems with cryptocurrencies, a user has to pay for the execution of a smart contract. This means a DoS attack would be so

expensive that it is not realistically feasible. The second problem for the order-execute architecture is *non-deterministic code*. The problem with non-determinism is that peers in a network may have different results when executing smart contracts, and thereby the blockchain will fork. Typically, blockchain systems that support smart contracts impose a domain-specific language (e.g., Solidity for Ethereum) to enforce determinism. The problem is that a programmer needs to learn a new domain-specific language for every blockchain system that supports smart contracts. But a general-purpose language like Java would introduce the thread of non-determinism (also by hidden implementations). Lastly, the *confidentiality of execution* can be a problem. The problem is that for some use cases in permissioned systems, not all peers should run all smart contracts. Thus, it would be possible to achieve confidentiality in smart contracts. However, the drawback of such an approach is a considerable overhead. A solution for this problem would be to run the smart contract on a set of trusted peers and then propagate the result to the other peers [ABB+18].

Figure 3.4 shows a Hyperledger Fabric network example. All peers maintain the blockchain ledger and thereby record all transactions. Clients send their transaction proposals to peers (indicated with letter P) according to an endorsement policy. An example of such an endorsement policy would be that three out of five peers would need to support a transaction to get passed. Specific peers then execute each transaction, and the output gets recorded. Transactions enter the ordering phase after their execution. An ordering service then uses a pluggable consensus protocol (currently Kafka, Solo, and Raft [Fab20]) to generate an ordered sequence of endorsed transactions grouped in blocks. These blocks then get broadcasted to all peers (optionally with gossip). Hyperledger orders transaction outputs combined with state dependencies computed during execution and not transaction inputs compared to most other blockchain systems. All peers then validate the transaction in the same order in a deterministic way.

Hyperledger Fabric proposes an execute-order-validate architecture. Figure 3.4 shows the different steps of this architecture. It consists out of four phases:

Execution phase In this phase, clients send and sign their transaction proposals to the peers defined by the endorsement policy. These peers then simulate the proposal by running the smart contract in a docker container against a peer's local blockchain (without persisting the state). Therefore, a peer can abort execution if it suspects malicious code (e.g., a DoS attack). The result of the simulation is a written set consisting of state updates and a read set consisting of version dependencies of the proposal. This result will then be cryptographically signed and sent back to the client.

Ordering phase If a client receives an endorsement of enough peers, the transaction gets assembled sent to the ordering service. Such a transaction consists out of smart contract code, parameters, metadata, and a set of endorsements. The ordering phase then creates a total ordering of all transactions per channel and atomically broadcasts endorsements. By doing so, it reaches a consensus on transactions. Additionally, the ordering service puts transactions into blocks.

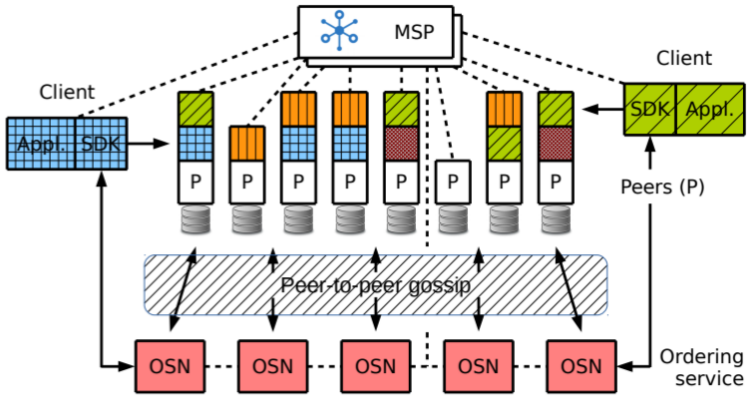

Figure 3.4.: Example of a Hyperledger Fabric network running multiple smart contracts (indicated by shaded and colored boxes) on peers according to a given endorsement policy. Taken from [ABB+18].

Validation phase The validation phase checks the fulfillment of the endorsement policies. These checks happen in parallel for all transactions within a block. Only designated administrators can modify these endorsement policies. Additionally, it checks for read-write conflicts. It checks if the conditions for a transaction are still valid on the current state of the ledger. Otherwise, it gets marked as invalid.

Update phase The update phase appends the new block to the local ledger, and the blockchain state gets updated.

The performance evaluation of Hyperledger Fabric has shown that it reaches under optimal conditions up to 3500 Transactions Per Second (TPS) with an average latency of 500 milliseconds for creating or spending a test UTXO cryptocurrency [ABB+18]. As the optimal condition is not realistic for real-world application, Figure 3.5 shows the results over multiple datacenters with 20 peers in each datacenter. We can see that we get a peak performance of around 1500 TPS with gossip in the data center with the lowest network performance. As Hyperledger also focuses on private or consortium blockchains, 20 peers give a good indication of its performance. Nevertheless, the comparison to a public blockchain can indicate the performance of a permissioned system. Bitcoin, for example, achieves a maximum of seven TPS and a latency of more than ten minutes [CDE+16]. Ethereum reaches up to 15 TPS [Eth19].

3.1.5. Blockchain Database Comparison

Before analyzing specific blockchain databases, we need to define some properties that help distinguish the different approaches. To find fair properties, we combine features which

	HK	ML	SD	OS
netperf to TK [Mbps]	240	98	108	54
peak MINT / SPEND throughput [tps] (without gossip)	1914 / 2048	1914 / 2048	1914 / 2048	1389 / 1838
peak MINT / SPEND throughput [tps] (with gossip)	2553 / 2762	2558 / 2763	2271 / 2409	1484 / 2013

Figure 3.5.: Performance test over multiple datacenters taken from [ABB+18]

the single approaches use to demonstrate their performance. According to Chowdhury et al. [CCK+18] (comparison to central databases) and the BigchainDB whitepaper [Gmb18] (comparison to distributed databases), the advantages of a blockchain are:

Decentralization A blockchain operates without any single governing authority. It works by having a group of nodes that make decisions and run the blockchain. A database typically needs a central trusted party.

Robustness/fault tolerance In a blockchain system, data gets distributed among nodes. Additionally, blockchain systems typically enforce Byzantine Fault Tolerance (BFT). BFT means that less than one-third of the nodes can fail in any way, and the system will still be able to agree on how to proceed. Centralized Databases store data in a centralized manner, not restricted to a single physical site (e.g., a single Server). Nevertheless, storing all data in a centralized way introduces a single point of failure (e.g., if an interface for storing is no longer reachable). As a result of a centralized database failing, the database is no longer available for applications.

Immutability In a blockchain system, data gets written in an append-only manner into blocks containing transactions. Once a transaction gets written, it cannot be deleted or modified. New transactions (e.g., by using smart contracts) can only modify the state. These changes will be recorded and are therefore traceable. In a typical database, data can always be changed or deleted.

Owner-controlled assets In a blockchain, only asset owners can transfer assets. An owner is a holder of a set of private keys. For example, it is not possible to transfer Bitcoin without a private key. Databases have support for user authentication, role, and group management. The main difference here is, that a user does not need to sign a transaction. Therefore, a transaction's sender is not stored in a persistent (and immutable) manner by default.

Redundancy In a blockchain system, each participating node has a copy of the latest data. However, in a centralized database, only the central party has a copy of the latest data. Furthermore, as not all applications need all data on-site, a data storage central location may also be beneficial due to storage (i.e., number of hard drives) and synchronization requirements.

Security Blockchain uses cryptographic measures by default. For example, transferring a cryptocurrency (e.g., Bitcoin) to another cryptocurrency wallet requires the private key of the owner. Thus, the transaction is signed by the owner and thereby secured. On the other hand, a database may use more traditional access control. For example, in a database, a representation of a coin may not require a digital signature of the owner to be spent.

The database advantages are:

Confidentiality of data Databases can restrict the access to certain data to authorized persons. Similarly, some permissioned blockchain systems support the notion of confidentiality [NGS+19]. Note that confidentiality is clashing with the concept of traceability by all participants. Therefore, it will only influence the evaluation if it is optional.

Performance A blockchain takes time to reach a consensus (e.g., around 10 minutes for Bitcoin). A database has a high transaction rate, high capacity, and low latency, which yields an immediate execution/update.

Indexing and querying Traditional blockchain systems do support only restricted indexing and querying. On the other hand, databases can support complex queries as well as a variety of indexing techniques. The lacking indexing and querying options become noticeable when searching for the content of a transaction (e.g., to find a result of the execution of a smart contract in a block).

Storage requirements As blockchains create a significant storage overhead (e.g., transaction metadata), storage is a concern. As a blockchain replicates the data over multiple nodes (either storing parts or all data), the data storage consumption is significantly higher than a centralized database. This criterion was not applied as the systems did not provide sufficient information to compare them on a storage basis.

The following criteria are not taken into account, as there are similar approaches in both blockchain and database systems:

Smart contracts Smart contracts are computer protocols that help to specify and enforce certain rules. As indicated by Nathan et al. [NGS+19] smart contracts are reproducible by stored procedures (e.g., PL/SQL procedures). The critical difference is that smart contracts are deterministic and stored procedures only under certain conditions.

First, we will evaluate the SQL-based blockchain database solutions, and then we will go over the blockchain-based blockchain database solutions.

BigChainDB [Gmb18]

BigChainDB is a blockchain database that imitates Blockchain storage. It uses Tendermint to achieve BFT consensus and has local MongoDB instances at each node.

Decentralization

The distributed nodes are running a local MongoDB database. Tendermint is connecting these nodes for networking and consensus.

BigChainDB: robustness/fault tolerance

The consensus network of BigchainDB, Tendermint, is BFT. The worst thing a malicious attacker with access to a local MongoDB instance can do is to delete or corrupt the local data.

BigChainDB: immutability

Every node has a full copy of all the data in a MongoDB database. Complete copies prevent the destruction or corruption of data. Additionally, every transaction is cryptographically signed. Therefore, changing the content of a transaction would be detectable.

BigChainDB: owner-controlled assets

BigchainDB implemented the concept of ownership based on private keys. It also supports multiple assets, while most blockchain systems support only one (e.g., Bitcoin).

BigChainDB: redundancy

Data is kept redundant, as each node has a copy of all data in their local MongoDB database.

BigChainDB: security

Firstly, BigchainDB used encryption and implemented a measure against double-spending to prevent spending an asset like bitcoin twice. Secondly, Sybil attacks which gain control over the network by having many nodes are impossible in their intended use case. Additionally, the authors [Gmb18] also assume that a governing organization behind the network controls the list of nodes. Therefore unknown, and untrusted parties are not allowed into the system.

BigChainDB: confidentiality of data

Confidentiality of data is partially given as the assumption of BigchainDB is that a governing organization controls the list of nodes. But everyone with access to a MongoDB database could read all data.

BigChainDB: performance

According to their bottleneck, the consensus network Tendermint [JK19] could process thousands of transactions in a test with 64 nodes in seven data centers on five continents.

BigChainDB: indexing and querying

Every node has its own local MongoDB database. MongoDB supports efficient indexing and querying of stored data.

IBMs approach [NGS+19]

The IBM *blockchain meets database* approach uses a modified version of SSI to ensure consistency and order. It uses PostgreSQL for storage.

IBMs approach: decentralization

Database nodes and ordering nodes are running in a distributed manner. Each database node maintains its (independent) copy of the ledger.

IBMs approach: robustness/fault tolerance

The ordering service is pluggable. Therefore a byzantine fault-tolerant consensus algorithm is usable. A modified SSI is running on each node independently to serializes transactions. A node atomically stores all information about transactions and their status in a table. If it fails during an operation, it recovers, when it comes back online.

IBMs approach: immutability

The system becomes immutable by storing the hash of a block and the previous blocks (or changes become at least detectable). Instead of deletion, the database uses the concept of flagging rows to indicate the deletion.

IBMs approach: owner-controlled assets

Each user has a digital certificate registered with all database peers in the system. The digital certificate is used to sign and submit transactions in the system.

IBMs approach: redundancy

Each database node runs its replica of the ledger. Nodes separately execute stored procedures, validate a block and commit blocks of transactions.

IBMs approach: security

Send and receive operations communicated over a secure communication protocol (such as TLS). Send and receive operations communicated over a secure communication protocol (such as TLS). Each node has a cryptographic identity. The identity of the node authenticates and signs all communication cryptographically. It also proposes ways to prevent distributed denial of service and *51% attacks*. In a 51% attack, the attacker tries to get 51% of the hashrate to control the network. Once the attacker has control over the network, double-spending becomes possible. Double-spending allows spending the same cryptocurrency more than once.

IBMs approach: confidentiality of data

The system does not handle access control to specific data directly. Stored procedures could include access control within directly.

IBMs approach: performance

The system does support parallelism during the execution of transactions and consensus to construct and order blocks. Asynchronous commits are serializable over multiple nodes. The system was able to execute around 1800 transactions per second on a test that included value insertions into a table. The test was conducted in a network using four data centers spread across four continents.

IBMs approach: indexing and querying

Each database node is based on PostgresSQL. By doing so, it enables indexing and querying. The system also supports join and aggregation operations. These operations are typically complex and expensive to implement for a blockchain-based system.

EthernityDB [HRIP18]

EthernityDB created a database structure in the form of smart contracts in the Ethereum blockchain. The smart contract architecture represents a document storage system similar to MongoDB.

EthernityDB: decentralization

Decentralization gets fulfilled due to the assumption that it has a direct monetary cost (e.g., Ethereum main network).

Robustness/fault tolerance

It is byzantine fault-tolerant due to its Ethereum basis.

EthernityDB: immutability

As the data is stored on-chain and the Ethereum blockchain is immutable, so is the data of EthernityDB.

EthernityDB: owner-controlled assets

As it runs on Ethereum, it supports Ether (similar to Bitcoin) as the currency of Ethereum.

EthernityDB: redundancy

It is redundant, if there are multiple full nodes (or even archive nodes). We can assume redundancy as the paper references the main Ethereum network.

EthernityDB: security

It provides the same strong security guarantees as the Ethereum network itself. Some attacks like the 51% attack would still be theoretically possible.

EthernityDB: confidentiality of data

According to the publicly available code, the concept of private databases exists only for the insertion of documents and collections, but not for reading it.

EthernityDB: performance

EthernityDB optimized the queries. But compared to a database, it still lacks throughput. According to the authors' theoretical calculation, the throughput lies around 281 bytes per second for insertions on the main Ethereum network. The JSON document's price with a 4 level depth was around 6€ (at a price point of 300€ per Ether) [HRIP18]. Although this sounds unreasonable, it counts only for the main Ethereum network. An option to make it more feasible would be a private or consortium blockchain network.

EthernityDB: indexing and querying

The system supports simple queries with exact matches and the "or" operator. It supports strings, timestamps, Booleans, and integer values. The representation as a string in Solidity (the programming language of Ethereum) is common, as it does not support certain types (e.g., floats). Textual strings can represent most datatypes.

SEBDB [ZZJ+19]

SEBDB is a blockchain database that uses a blockchain as storage while also allowing off-chain data storage in traditional databases. In addition, their approach supports sophisticated join and trace operations.

SEBDB: decentralization

The on-chain data is decentralized. A database management system handles the off-chain data with no indicator for decentralization.

Robustness/fault tolerance

They use a plug-in pattern supporting multiple consensus protocols. Among these is the byzantine fault-tolerant consensus protocol PBFT.

SEBDB: immutability

The data stored on-chain is immutable. No information indicates that off-chain is immutable.

SEBDB: owner-controlled assets

The test system runs on Starchain [Sta18], which has owner-controlled assets according to their whitepaper.

SEBDB: redundancy

They use heavy nodes (storing all data) and thin clients. By doing so, on-chain data is kept redundant. There is no indication that off-chain information is kept redundant.

SEBDB: security

The system supports authenticated queries restricting access to the system. As shown in the confidentiality point privacy, private data should remain off-chain. By that, a malicious participant or attacker that gets access to the blockchain or a single organization will not have all the sensitive data.

SEBDB: confidentiality of data

The idea of this system is that private information should remain off-chain. On-chain data is the only information needed by multiple participants. Figure 3.3 shows an example of a donation application using SEBDB. Donor, project, organization, and donee id are visible for all participants of the blockchain network. By that, organizations can transfer a digital representation of money. More private information such as the e-mail address is gets stored by the organization responsible for it.

SEBDB: performance

Fast access to certain blocks or transactions is possible due to indices. Also, in comparison to the EthernityDB, the indexing happens outside of the blockchain. By doing so, it avoids additional overhead and is much more flexible. But the system does not provide a

solution for the slow writing speed on the blockchain, even though off-chain storage may help reduce the data that needs to be on-chain. With this approach, off-chain data is not guaranteed to be immutable and stored redundant or fault-tolerant manner.

SEBDB: indexing and querying

Support of create, insert, select, join, and trace statements exist. Join statements enable joining different sources, i.e., off-chain with on-chain storage. Trace statements are exciting due to the immutability property of a blockchain.

3.1.6. Blockchain Database Comparison Overview

Table 3.1 shows a direct comparison of all presented systems. We exclude performance and security criteria due to missing information for a direct comparison. The symbol ✓ stands for fulfilled and ✗ for not fulfilled regarding the requirements. The text *only on-chain* describes that a property hold on-chain but not off-chain. Additionally, *only on-write* describes that the property holds for writing to the blockchain but not for other operations (i.e., reading). Furthermore, *only simple operations* describes that not all operations of a typical CRUD Database were implemented. SQL-based systems were successful in reproducing blockchain properties and vice versa. Although, all systems can improve at least one property. The presented methods have shown that stored procedures can replace smart contracts in non-blockchain systems. SEBDB has shown that with the traditional blockchain system, a viable solution in terms of performance may be to store as little as possible in the blockchain. While EthernityDB came to a rather negative conclusion on using blockchain storage in Ethereum, SEBDB achieved a good performance while reading. While EthernityDB defines a query engine as a smart contract, SEBDB stores data into the blockchain and queries it via indices.

Nevertheless, for these systems, on-chain writing performance will remain a problem. Hyperledger Fabric, on the other side, achieves under certain conditions much better-writing performance. Its architecture does not force all peers to execute all smart contracts and allows some parallelism. The comparison has shown that at the moment, only SQL-based blockchain database systems can achieve good results in all comparison criteria. In contrast, a combination of Hyperledger Fabric performance enhancements and SEBDB querying approach could be the key to a blockchain database with a good performance. By doing so, the resulting system's writing performance could be comparable to the "blockchain meets database" approach of IBM, which bases on PostgreSQL [NGS+19].

3.2. Related Work

In this section, we will discuss related work. We used the Digital Bibliography & Library Project (DBLP) [dbl21] search engine to find related scientific work. Additionally, we searched for existing practical projects and libraries. We restricted the search to mapping mechanisms and automated smart contract code generation. While every blockchain database discussed approach in the previous chapter is somehow related work,

	BigChainDB	IBM	EthernityDB	SEBDB
Decentralization	✓	✓	✓	only on-chain
Robustness/fault tolerance	✓	✓	✓	only on-chain
Immutability	✓	✓	✓	only on-chain
Owner-controlled assets	✓	✓	✓	only on-chain
Redundancy	✓	✓	✓	only on-chain
Confidentiality of data	✗	✗	only on-write	✓
Indexing and querying	✓	✓	only simple operations	✓

Table 3.1.: Comparison of different blockchain database approaches

the fundamental difference is that mapping is not their goal. An exception to this is Hyperledger Fabric, which supports a notion of mapping in their chaincode programming language (smart contract) files [Fab21]. The Hyperledger Fabric shim, a library that changes passed arguments from source code annotations to Hyperledger parameters for Java classes, allows the communication to Hyperledger Fabric peers. Although their purpose is different, these annotations work similarly to the Java Persistence API (JPA) annotations used in traditional ORM. Another difference is that this mapping occurs in the chaincode. Defining it in the chaincode means a developer needs to create a chaincode project instead of using his Plain Old Java Object (POJO)s directly. Therefore we do not categorize this approach as a ORM approach.

3.2.1. BigchainDB ORM Framework

The js-driver-orm project is a BigchainDB ORM framework. It follows a *Create Retrieve Append Burn* (CRAB) approach. The *Create* operation generates an asset or a token (by minting), *Retrieve* is capable of retrieving assets from the data source, *Append* adds a new state of an asset, and *Burn* transfers assets to a designated deletion public key. This approach differs from the traditional CRUD approach, as a blockchain does not support deletion or update in the conventional sense. As immutability means that we can't alter or delete old states, we need to append a new state, if we want to update a model representation on-chain. The append operation also causes a transaction of the asset to the same owner. For the deletion, there is the same problem of immutability. As a result, a unique burn address is used (to mark assets as deleted) [Pre17, JP18].

While supporting the mapping of single objects, we could not find evidence in the test classes or documentation (Github [JP18] and article [Pre17]) that relations between different objects are supported within the ORM framework. Nevertheless, one could argue, that foreign keys could be stored as attributes. While foreign keys are representable as

simple attributes, it would require some logic on the user side to connect the models.

3.2.2. Ethereum ORM Framework

EthAir Ballons allows persistent storage in a model-oriented way. It works by generating contracts for user models by using a template. Similar to the BigchainDB ORM approach, it supports a notion of CRUD operations. The difference is that the BigchainDB ORM transfers deleted assets to a designated delete public key. EthAir Ballons, on the other hand, has a table contract containing all instances of a given model. This table contract holds an array. Upon deletion, the last entry swaps places with the to-be-deleted model. Additionally, the number of records (variable to store array size) gets reduced by one, and the last entry gets removed from the array [AK21b]. All transactions are on-chain, and the deleted model's smart contract still exists somewhere in the blockchain. While it does not break immutability, it will not be easy to trace these deleted entries [Dem20]. We could not find evidence in the test contracts or documentation (Github [AK21a] and article [Dem20]) that relations between different objects are possible within the ORM framework. While foreign keys are representable as simple attributes, it would require some logic on the user side to connect the models.

3.2.3. Domain-Specific Language Mapping

Frantz et al. [FN16] describe in their paper *From Institutions to Code: Towards Automated Generation of Smart Contracts* an approach to define institutional contracts in the form of Domain Specific Language (DSL). These contracts then get mapped to Solidity smart contracts. Before deploying them on Ethereum, they need some manual refinement. While manual improvement of generated code should not be encouraged in code generation [Fow02], their approach is still promising for other use cases. The problem with manually refining smart contracts is that, typically, changes get overwritten. Additionally, it creates a barrier when updating source code or the dependency version changes (as not within the system). For some use cases, it is essential to make assumptions to create complete smart contracts able to work without manual refinement. One of these use cases could be using a DSL mapping in the context of ORM.

3.2.4. Legal Smart Contract Generation

Hu et al. [HZD+20] proposed a framework to close the gap between smart contracts and legal contracts. Their framework focuses on verifying the models used for code generation. The code generation approach uses a Model Driven Architecture (MDA). At first, a formal language defines models which will get validated. Conversion rules enable the mapping from model to target platform code (e.g., Solidity code, if the platform is Ethereum). Additionally, it uses templates based on the conversion rules and a meta-model analysis. The required information for code generation gets extracted from the models by using a model analysis technique. A conversion engine then converts the information to the target platform code. In the next step, a runtime verification method validates the generated

code. As their case study only contains the contract verification part of their framework, it is unclear whether they implemented a prototype of their code generation process.

3.2.5. Usage of Natural Language Processing in Code Generation

Monteiro et al. [MRK+21] propose the usage of NLP to create smart contracts based on legal documents. First, legal documents get added in the form of text formats. These documents will then get preprocessed (e.g., by applying stemming, removing stop words). Basic smart contract classes then get generated with the preprocessing results. These results then get used in templates encoded with the smart contract grammar. Their prototype replaced template class keywords (e.g., Template) with the NLP result words. The paper's focus is mainly on NLP and less on a code generation architecture, as the paper never describes the structure of the generated classes. Another paper that uses NLP for code generation was published by Tateishi et al. [TYSS19]. Similar to the other approach, they use templates that run on Hyperledger Fabric.

3.2.6. Smart Contract Generation to Ensure Internet of Things Privacy

Loukil et al. [LGGBB18] provided a framework to generate smart contracts which manage privacy settings in an Internet of Things (IoT) environment. Their proposed Semantic IoT Gateway consists of a semantic rule manager that generates a privacy policy matching the owner's privacy policies and the data consumers' needs. Additionally, a smart contract factory creates smart contracts based on the privacy policy. It generates smart contracts, that manage privacy permission settings. These define the permission of IoT resources to store data locally, to send data to external storage, to read from other IoT resources, to write to other IoT resources, and to monitor (receive periodic data) from other resources. Additionally, it creates a smart contract that regulates ownership. It enables the addition of new IoT resources, their modification, and their removal. Finally, it creates a privacy policy smart contract, allowing the owner to share the output of IoT resources with consumers by providing add, remove, and update functions. A lightweight messaging protocol gets the terms of service of the customer and allows posting smart contract parameters. At last, a blockchain client provides an access point to the blockchain.

3.2.7. Usage of Domain-Specific Ontologies to Generate Smart Contracts

Choudhury et al. [CRS+18] proposed an approach that maps semantic rules and domain-specific ontologies to smart contracts. They defined the semantic rules in the appropriate grammar. For code generation, a smart contract template is used. The template is then converted into Abstract Syntax Tree (AST) representation. Constraint variables are then searched within the AST representation and updated with the appropriate values (given from the rules). The resulting AST will then be converted back to source code.

	Ready to use	General purpose	Code generation framework
BigchainDB ORM	✓	✓	N/A
EthAir Ballons	✓	✓	Text replacement in templates
From Institutions to Code	✗	✗	Templating using DSL
Legal Smart Contract	✗	✗	Templating using MDA
Natural Language Processing	✗	✗	Templating with NLP
Internet of Things Privacy	✗	✗	Code generation using factory pattern
Domain-Specific Smart Contracts	✓	✓	Templating using a AST

Table 3.2.: Comparison of the related work

3.2.8. Categorization of Related Work

Table 3.2 shows a direct comparison of all related work. The criteria for this comparison are

- **Ready-to-use** describes if the outcome of the commercial or research project is publicly available and usable on different use cases.
- **General purpose** describes if the commercial or research project is restricts itself to a specific use case.
- **Code generation framework** describes how the system uses code generation (i.e., using a template option).

✓stands for fulfilled and ✗ for not fulfilled regarding the requirements. N/A means it is not clear from the paper or official website. It is visible from Table 3.2 that depending on the use case, there exist multiple ways to generate code. There are also some general-purpose solutions available. However, the ready-to-use systems we have seen support only one blockchain system.

4. SmartCOM Architecture

In Chapter 2 we have introduced the basic properties for SmartCOM, then we have looked into state-of-the-art projects described in Chapter 3. This chapter discusses the SmartCOM's general architecture without getting into detail with the different blockchain providers. First, we will give an overview of the various components. Subsequently, we describe the components in detail. Finally, Chapter 5 describes the implementation of the concepts discussed in this chapter.

4.1. Overview

Figure 4.1 shows the component diagram of SmartCOM. It consists of a user project which contains annotated user files. These files then get processed by the processor component of SmartCOM. The processor component extracts the necessary information in a format given by the target API. The target API then sends the information to the user-selected provider and triggers the smart contract and data repository creation. The user's dependencies select the blockchain provider. Each provider implementation uses APIs provided by the specific blockchain ecosystem to interact with the blockchain itself (i.e., when deploying or invoking smart contracts or transactions). The providers also copy the necessary smart contracts as well as the data repositories into the demo project. By doing so, the user can interact at runtime with all the required classes. The management of the blockchain is outside of the scope of our system. By management, we mean that the blockchain needs to be started and configured before our application can use it. Also, user creation (e.g. creation of wallets) must happen outside of our application. Credentials allow the linking of users to the user project application (e.g., entering username/password). Figure 4.2 describes a possible (i.e., the ordering of the execution can vary) interaction between a developer and SmartCOM in detail. In order to use SmartCOM it needs to be included as dependency into the user project (by the developer). As soon as all POJOs are annotated the code generation process can start. After compiling the source code the generated data reposiories, Java connectors, and converter classes can be found in the user project. If the blockchain is started and configured the developer can use his user project directly with the blockchain system.

4.2. User Project

The user project needs to be a project written in the Java programming language. The project needs to contain Plain Old Java Objects (POJOs), represented as database tables. POJOs are Java classes that should not extend, implement, or include annotations of

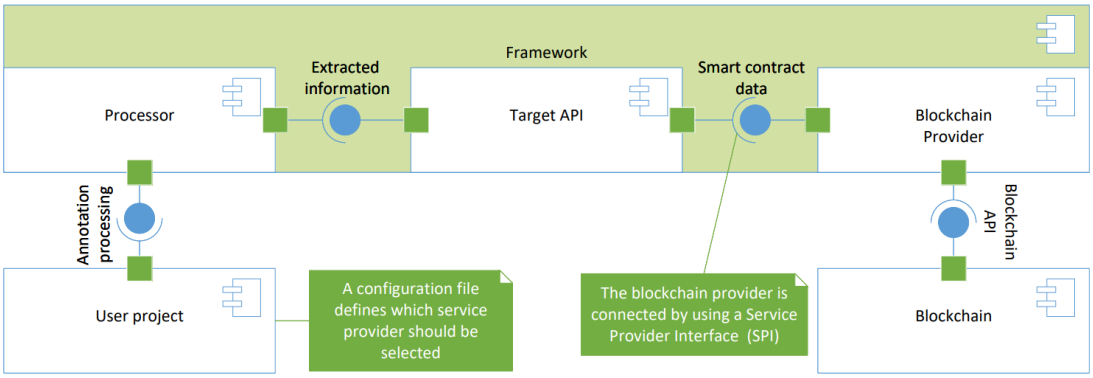

Figure 4.1.: Component diagram of SmartCOM

prespecified classes. Figure 4.3 shows the conversion of a POJO to a database table filled with three sample entries. To convert POJOs to database tables, they need to contain metadata information. SmartCOM uses syntactic metadata in the form of annotations. The user selects the blockchain system used for storage by adding a dependency on the blockchain provider component of his choice and a blockchain information configuration (including host, port, etc.). Additionally, a velocity configuration file defines where generated classes should be stored and serves as information for the service provider interface that the blockchain provider should use.

4.2.1. SmartCOM Annotations

SmartCOM currently uses six types of annotations. The user can use these annotations to persist his data on the blockchain system of his choice.

@SmartContractProperty The smart contract property specifies that SmartCOM should process a user-defined POJO. If this annotation is declared, smart contracts and data repositories get created. The annotation is comparable to the JPA @Entity annotation. The goal of both annotations is to create a table with given attributes (or columns). The tables get named according to the class name.

@AttributeProperty Attribute property specifies that the annotated Java attribute gets converted to a smart contract attribute (or column). This annotation is similar to the @Column annotation of the JPA. The important aspects are that the modifier, type, and name should be correct as converted by SmartCOM. If the modifier is public, then reads and updates are directly applied. Otherwise, we assume that a getter and setter exist. An attribute gets converted either directly if an equivalent type exists or converted to a similar type. An example of such a conversion is floating-point numbers in Ethereum. Floating-point numbers are not yet usable in

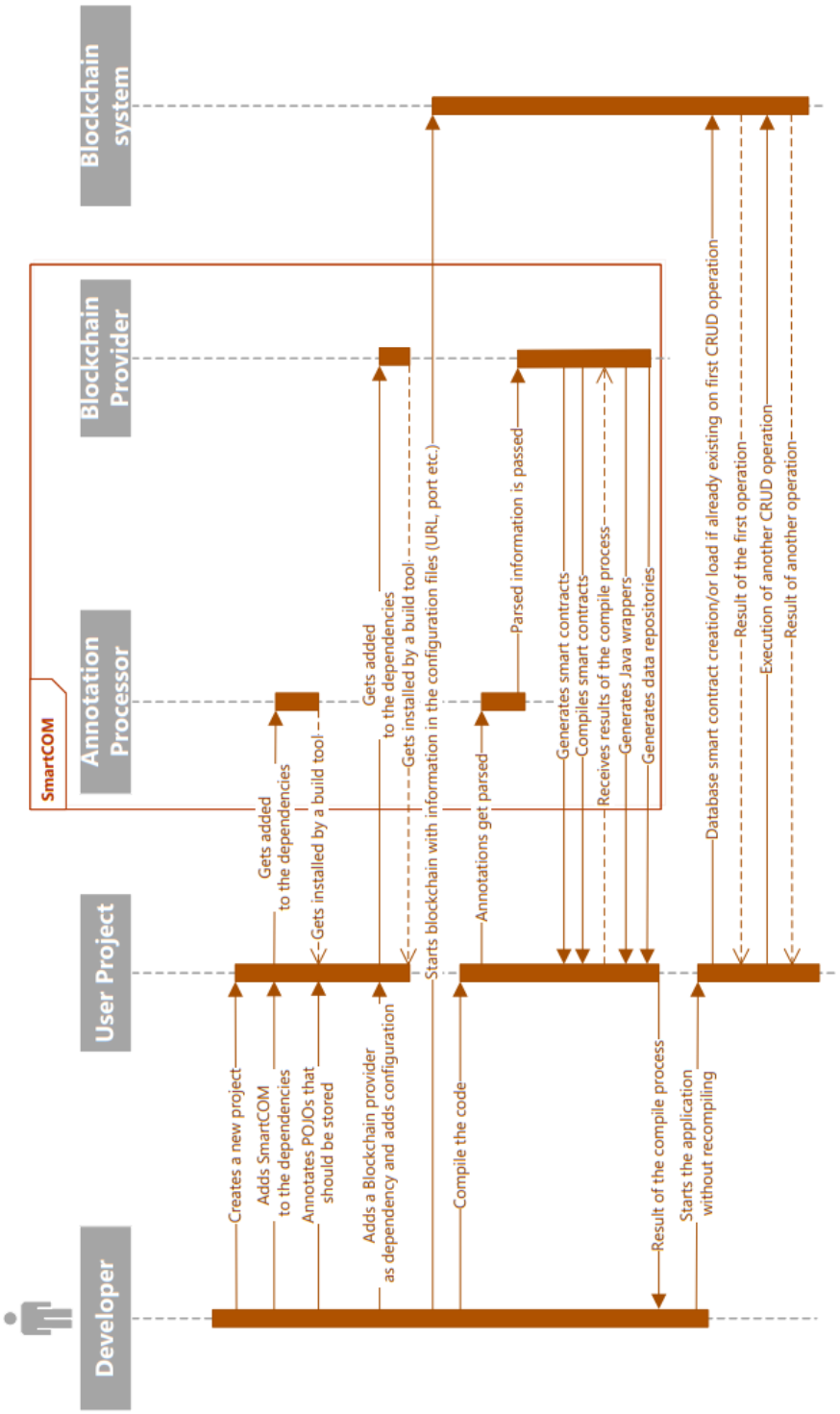

Figure 4.2.: Sequence diagram of SmartCOM

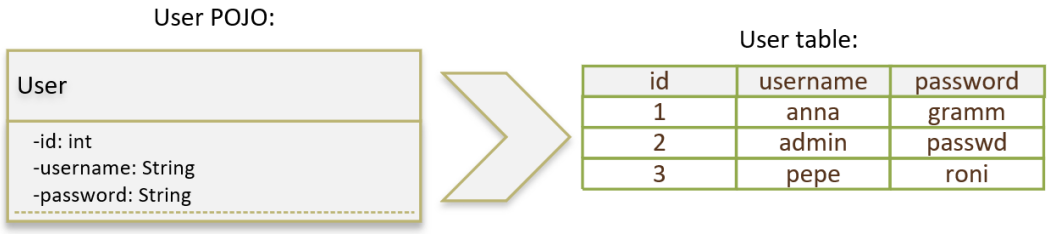

Figure 4.3.: Conversion of a user defined POJO to a database table

Solidity [Sol20]. Therefore, SmartCOM stores them as text and converts them back. A different approach could be to implement floating-point numbers by ourselves. We use the name of the attribute directly as the smart contract attribute name.

@OneToOneProperty A one-to-one annotation specifies the relationship of a single annotated POJO to another annotated POJO (1-to-1 relationship). This annotation is similar to the JPA @OneToOne annotation. The annotation does *not* require to be bi-directional. Bi-directional would mean, that if we have a relation from a user to an address, we would need to have it stored the other way around (from address to a user).

@ManyToOneProperty The many-to-one annotation specifies the relationship of multiple instances of an annotated POJO to a single annotated POJO (n-to-1 relationship). This annotation is similar to the JPA @ManyToOne annotation. This annotation is not required to be bi-directional.

@OneToManyProperty The one-to-many annotation specifies the relationship of a single annotated POJO with multiple instances of an annotated POJO (1-to-n relationship). This annotation is similar to the JPA @OneToMany annotation. This annotation is not required to be bi-directional.

@ManyToManyProperty The many-to-many annotation specifies the relationship of multiple instances of an annotated POJO with multiple instances of an annotated POJO (n-to-n relationship). This annotation is similar to the JPA @ManyToMany annotation. This annotation is not required to be bi-directional.

4.2.2. Configuration

The user needs to configure and start the blockchain system by himself. Nevertheless, some information is needed to know where to find and interact with the blockchain and the user project. The configuration files get defined at the user project in the resources/config folder.

Blockchain configuration

This file contains the information needed to connect to the blockchain. The entries depend on the blockchain system and include the Uniform Resource Locator (URL) and the port of an interface of a running Blockchain system.

Velocity configuration

The velocity configuration file includes the path to the selected blockchain provider implementation. The path corresponds to the Java class location (including the package) of the chosen provider. The Service Provider Interface (SPI) requires the path as it selects the provider according to this information. Additionally, the project name is needed as generated files will be placed in the user project's target folder (if possible).

4.3. Processor

This component processes the annotations in the user-defined POJOs. Additionally, it formats them in the target API format and passes them to the target API.

Annotation processing

The annotation processor parses the source files annotated with specific annotations. For example, by annotating a POJO with the *@SmartContractProperty*, our processor will process the POJO with all other annotations within the same file. The annotations will then be processed and converted to the format defined by the target API. Thus, the annotation processor separately records all annotated files, the imports needed by every class (due to relations between classes), all annotated attributes, and all kinds of relationships (e. g., one-to-one).

Translation file

If the smart contract language varies from Java, it is needed to provide a translation file. The translation file needs to be called targetTranslation.json. This translation file needs to be in JSON format as illustrated in listing 4.1. The datatype gets matched case insensitive and converted to the default declaration and method type if not stated otherwise. For example, if we want to support a timestamp datatype, an option would be to declare, that it should be a text string. Defining unsupported datatypes as text string will suffice, if the *toString* method is present (in the case of primitives, it uses string concatenation). If the conversion is more complex, then the developer needs to consider the type in the Velocity Template Language (VTL) files. As a result, there would be a check, if an attribute type equals *Date*, and then it converts it to a *Date* representation in the target language with the developer's code. We did not include this in the translation file as most types are representable by strings.

```
{"STRING": {
    "defaultDeclaration": "string",
    "defaultMethod": "string memory"
},
"INT": {
    "defaultDeclaration": "uint",
    "defaultMethod": "uint",
    "alternatives": ["int"],
    "alternativeMethods": ["int"]
}}
```

Listing 4.1: JSON translation file

Connection to the blockchain providers

The processor component calls the methods offered by the *target API*, which interact with the SPI to generate the smart contracts and the data repositories. The implementation details are hidden from the processor component, as it only knows the signature of the methods. Therefore, information gathered from processing annotations can be passed to the blockchain providers implementing the service provider interface.

4.4. Target API

The target API handles all providers' components and defines the interfaces, that need to be extended by the providers. Therefore, it represents the service provider interface. It also describes the format in which the processor passes the information to the different providers. An example is how attributes get stored and which information is required to create smart contracts.

4.4.1. Service Provider Interface

Some methods need to get implemented to guarantee, that all blockchain providers implement the minimal requirements to create a blockchain-based database. If a specific blockchain provider does not require a step, then it can be left empty. The following four steps (methods) are required.

1. **Project initialization**. In this step, necessary predefined files or directories get copied. For example, these can be template files implemented by the different generated smart contracts, libraries, or build files to build a smart contract project. In this step, the blockchain provider gets no information about the user project as it should only prepare the code generation process.

2. **Write instance file**. The information gathered from a single annotated POJO gets used to create a smart contract instance file in this stage. If other smart contracts

are needed to create a similar structure as a table, they will get created. In this stage, the blockchain provider gets information about the POJO converted to a smart contract and general information regarding all other POJOs (e.g., due to imports when there are exist relationships).

3. **Instance processing**. In this stage, all smart contract instance files get processed. Processing may be done via a build file or directly compiled with a compiler. This stage will execute after all instance files have all possible dependencies ready. This method does not include parameters and exists solely to compile files and delete artifacts that are no longer needed.

4. **Write repository**. This stage will create Java data repositories that follow a uniform interface to create, read, update, and delete the data stored on the blockchain component. As a result of this, the user can rely on the uniform interface which is using his POJOs. When switching between different blockchain providers, only the parameter of the user management can change. We need other parameters as different blockchain systems may use various security measures. Nevertheless, the uniform interface will not change. This stage runs after the instance processing. The repository may depend on the processed instance (i.e., when Java connectors get generated). In this stage, the blockchain provider gets information about the POJO converted to a smart contract. Additionally, it gets general information regarding all other POJOs for the case that there are other relations.

Additionally, the service provider also defines some utility functions needed by all providers. For example, it provides some file creation utilities and velocity engine configuration. As all blockchain providers implement the SPI it reduces the amount of code duplication.

4.4.2. Repository Interface

A repository interface is needed to guarantee the same basic methods across all blockchain provider implementations. This interface enables the user to switch between different blockchain providers without making significant changes to his code. Several methods need to be implemented by a blockchain provider data repository. These repositories define Create Read Update Delete (CRUD) methods, that use the user-defined POJO. Additionally, the repositories have some blockchain-related functionality (user management and a trace method). The interface includes the following methods:

Create Upon passing the create method, a POJO a new smart contract representing the POJO gets created on the blockchain. This method then generates an instance of a smart contract and stores it on the blockchain. A POJO with an empty identifier (id = 0) gets passed to the create method to create an instance. The POJO's id will then receive an autogenerated value and get returned. If the creation of the smart contract fails, an empty value (null) gets returned. Upon creation, all details of an instance, a timestamp, and the creating user get recorded.

Read find one eager The find one method returns a POJO instance by its autogenerated identifier (id). Find one searches the smart contract with the identifier on the blockchain. If this method does not find a matching smart contract (if not marked as deleted), the method returns an empty value (null). Otherwise, the smart contract gets converted to a POJO and returned.

Read find all eager To find all instances of a given table, the method *find all* exists. This method returns all not deleted instances of a table as a collection. It checks the count of a given table and calls find one on all instances.

Read find one/find all deleted eager These methods work in the same way as finding one/find all eager except that they also return deleted entries (useful for traceability).

Read (find one, find all) semi-eager In comparison to the eager variant, no connected instances get loaded. The only information SmartCOM retrieves from related instances is their id. SmartCOM users require the id to load connections by their id in a semi-eager way. The advantage of semi-eager loading is, that it is much faster than the eager variant. Similar to the eager variant, we can define, if we want to load deleted instances.

Update Upon receiving a POJO from the user application, SmartCOM checks the autogenerated identifier of the POJO exists on the blockchain. If the method receives an existing instance, it will update a smart contract on the blockchain. This method returns the resulting POJO after the update or an empty value (null), if no instance gets found on the blockchain. Each update results in recording the changes, the time of change, and the user who changed it.

Delete A POJO is marked as deleted when passing the auto-generated identifier (id) of a POJO to the delete method. As a blockchain system is immutable, delete has a different meaning. Deleting in SmartCOM means marking an entity as deleted and logging when and which user deleted it.

Trace Trace finds all trace entries of a given smart contract on the blockchain by its identifier (id). A trace entry consists of the eager POJO representation of the smart contract. The eager representation also contains all connections and their connected instances. Additionally, a timestamp and which sender has committed the trace is stored.

Set user This operation changes the current session user to a user that implements the session user interface. This interface requires an identifier. The session user may not be the same for different providers as the authentication process varies between them. The only thing that is the same over all tested blockchain systems is that they need at least some string identifier. The users themselves need to get created outside of SmartCOM. The set user method connects the blockchain user (e.g., entry in a wallet) with the application.

4.4.3. Predefined Models

To guarantee some properties and to ease the use of SmartCOM, we defined some predefined models. Some of them need to get implemented, while others are simply a generic format.

Abstract model

To guarantee that a user-defined POJO has all attributes needed to trace an object over time, it needs to extend an abstract model. This model has three attributes: an identifier, a version, and a deleted flag. The identifier is required to identify a smart contract on the blockchain uniquely. This identifier can be used for updating, finding, and deleting an instance. The version flag gets used to connect traced smart contracts with the correct version at the change time. The delete flag is true if an entry is deleted and false otherwise. This flag is essential to the find one/find all deleted methods as they also return deleted instances. Additionally, a user-defined POJO should implement a hash code based on its attributes and the relations identifiers. It should not include the relationships themselves as hash code gets used to check, if changes occurred on a single instance and relationship cycle prevention.

Session user

The session user represents all information needed to change the current application user to a new application user known by the blockchain system. If additional attributes are required, then the session user gets extended. As SmartCOM does not provide user creation capabilities on the blockchain per default, it is vital, that the user already exists and is known to the blockchain system. The session user then can be used for the set user method in the data repository.

Attributes

For attributes, name, type, original type, method type, and modifier get recorded. The name is the field name of an attribute. The type is the field type converted to a type in the smart contract language of the provider. In the case of generic types, the interface parameter is used (e.g., a list of users would result in a user class). This is important because we already know whether to create a single entity or a list due to the annotation. This makes it easier to convert it to the smart contract language. The original type is the entire field type. The original type is important to know when a smart contract instance gets converted back to a POJO. The method type is only important for some smart contract languages as the modifiers required for methods may not be the same as for attribute declaration.

Field

The field model is used to unify the representations of attributes in velocity. It can be extended by the blockchain providers when needed. Additionally, the attribute information contains the getter function name, setter function name, and original type (required for conversion to Java).

Trace entry

A trace entry consists of three attributes: a user-defined POJO model given as a generic parameter, a String instant representing the time instant of the trace, and a String sender representing the user responsible for the trace. The model represents the POJO state and all related POJOs at the time of the change. The instant is a time representation of the Java datatype instant, converted to a string. The sender information is also stored as a string, as it is different for blockchain providers. Nevertheless, the sender can be represented as a string and should give the necessary information.

Type translation

The type translation gets defined in the target API. It uses the target translation file of the blockchain provider, which the processor reads. If no translation is needed, no translation file needs to get created and every type gets used as defined. Otherwise, types translate to their default value in the translation file. Alternative types are present for future use. The best option would be to include a parameter for the annotations that let the user select one alternative defined in the translation file (to guarantee the compatibility of data type and the conversion).

Utility methods

The utility methods include a system utility, a YAML Ain't Markup Language (YAML) loader, and a file utility. The system utilizes an installed operating system and can run commands directly in your system's terminal/command line. Command execution is important for external software (such as smart contract compilers) with no Java API. The YAML loader parses a YAML file into a map of string pairs. YAML is necessary for the project information given by the user project's configuration. The file utility can copy files to an arbitrary folder or the target folder.

4.4.4. How to Add Blockchain Providers

In this section, we discuss the requirements of blockchain providers and how to add them. Chapter 5 includes two concrete examples of such providers.

Blockchain Storage Options

A blockchain system is required to provide the possibility to store data (at least as strings) to be used in SmartCOM. Furthermore, some kind of smart contract would be helpful

even though it would work with key-value storage. Subsequently, the blockchain needs to allow state transition between different contracts/files to be traceable (instead of creating new unrelated contracts). An additional requirement is that the blockchain needs to provide a way to find historic states without scanning every transaction in the blockchain. Observing the blockchain for changes or storing the changes off-chain is not a feasible option. It is not always possible to guarantee that an application is running from the start of the blockchain or the integrity of off-chain data. Although scanning the blockchain to get the initial state would be possible.

Interaction with a Blockchain

SmartCOM requires a remote connection to interact with a blockchain system (e.g., over a REST API). There exist many Java RPC APIs for different blockchains. Furthermore, the maximum transaction size needs to be large enough for at least a single instance of the largest POJO the user might want to store. Subsequently, a table structure must be representable by separate on-chain contracts/files (e.g., through having relations to other contracts). Additionally, the contracts are required to be backward convertible to a POJO. Therefore, also using a hashing or encryption function would be an option if uniquely invertible. Finally, recording the sender and time of a transaction, when sending the transaction on the blockchain side is important. It is vital as if passed as a parameter, someone could post in the name of a different sender or alter the time.

Blockchain Deployment

As blockchains have a write performance problem (especially considering latency), keeping the number of nodes as small as possible is vital. Therefore, a private or consortium deployment is required. Another reason is that in a public environment, transactions typically have a direct monetary cost. Furthermore, a requirement is that the single transaction is required to have no direct monetary costs depending on a cryptocurrency market. Without the power over the network, it can get too expensive for practical usage. Additionally, for privacy and environmental reasons (e.g., higher power consumption, need for additional hardware), it is required to store an organization's data on a private or consortium blockchain instead of a public blockchain. Lastly, the blockchain (and typically a miner) needs to be active when an application could send a transaction. An active blockchain is needed, as there is no transaction queue in the current state of SmartCOM. Finally, the deployment of new contracts on the blockchain needs to be possible at runtime. Contracts for new applications are then addable to the same blockchain deployment.

4.5. Blockchain Provider Integration

In this section, we explain how a new blockchain provider, supporting a new blockchain, is added to SmartCOM. Every blockchain provider is an implementation of our Service Provider Interface (SPI) using the target API. A blockchain provider implementation is a Maven (a dependency and build tool) module that gets included when needed. The

easiest way to start is to copy the write provider files (from another blockchain provider under at.ac.univie.target.impl) and create a folder for the Velocity Template Language (VTL) templates in the resources. Write provider files handle the writing of files (i.e., which keywords to replace with parsed information). At first, it is best to create a simple class which is using some of the parsed information to understand the VTL syntax. Then, after generating simple contracts from single POJOs, a developer should generate data repositories to interact with POJOs. At last, after testing, a developer should implement complex operations such as handling relationships to other POJOs. As code gets generated at compile-time and tests are executed at runtime, most debug tools won't find breakpoints in the newly generated code. Therefore, logging functions are needed to report the state of operations. Another option for debugging is to remove the compile phase before testing (to prevent code generation). Often logging functions or verbose mode of the blockchain can help with development (e.g., docker output in Hyperledger Fabric). The user project must include the blockchain provider Maven module in the dependencies to use the blockchain provider. Additionally, the configuration file must contain the package name of the SPI (e.g., at.ac.univie.target.impl.HyperledgerWriteProvider) as well as all information to connect to the Blockchain (i.e., URL and port).

5. Proof of Concept

After looking at the general SmartCOM architecture in Chapter 4, we will now discuss the prototypical implementation. As seen in the last chapter, the Smart Contract Object Mapping (SmartCOM) architecture expects the different blockchain providers to use interfaces. Therefore, the blockchain provider implementation is up to the developer's needs. At first, we will discuss the technology stack shared among blockchain providers. Then we go into detail with the two showcase blockchain providers Hyperledger Fabric and Solidity. We will discuss blockchain-specific required software for each showcase and the blockchain provider's proposed architecture. To test the system, we provide a simple REST API able to execute operations when given POJOs (adding or updating) or their *ID* (deleting, reading, or tracing). At last, we will discuss the performance of our system and compare it to blockchain database systems.

5.1. Technology Stack

In this section, we will discuss the technologies and APIs used for the prototypical implementation of SmartCOM and possible implementation variations.

5.1.1. Apache Velocity

Apache Velocity is a free Java-based open-source templating engine. The templating engine provides the Velocity Template Language (VTL), which allows the definition of templates. We selected Velocity as it does not restrict the usage exclusively to web design. VTL is similar to Java Server Pages (JSPs). It allows referencing methods defined in Java code. It also supports conditions (i.e., if and else), loops, and including code of other VTL files (to reduce code duplication). Reduction of code duplication is essential to keep the code simple, maintainable, and less redundant. Such directives start with a hashtag (#). The language replicates the intends except when they are in the same line as directives. Listing 5.2 shows a VTL example that creates a Java class with package *packageName*, class name *className*, and all attributes stored in attributes except when they have the type *HashSet*. The variables packageName, className, and attributes need to be defined when processing the VTL file. Listing 5.3 shows the result with the parameters of listing 5.1. For example, it can be seen that the *$className* in listing 5.3 was simply replace by the *className* shown in listing 5.1, while the field name *username* was not written due to the *if* condition.

```
classname : "User"
attributes :
              - attributeType : "String"
                fieldName : "username"
              - attributeType : "HashSet"
                fieldName : "institutions"
```

Listing 5.1: VTL parameters

```

public class ${className} {
#foreach( $attribute in $attributes )
 #if(${attribute.fieldType} != "HashSet")
  private ${attribute.fieldType} ${attribute.fieldName};
 #end
#end
}
```

Listing 5.2: VTL definition example

```
package at.ac.univie.models;

public class User {

  private String username;

}
```

Listing 5.3: VTL result example

5.1.2. Service Provider Interface

The Service Provider Interface (SPI) allows discovering and loading of implementations of a given interface. These interfaces can be created by third parties and included in SmartCOM. In the case of SmartCOM, the interface defines methods to create smart contracts and data repositories. Data repositories allow create, read, update, and delete (CRUD) access to the blockchain's data. By using the SPI, different blockchain systems can implement the service provider interface. This allows switching between them without making changes to the code accessing the service provider interface. Another advantage is that the user project needs to include only the code for the blockchain provider of his choice. Usually, it would suffice to have a list of all providers in the META-INF/services directory of the project. Still, as SmartCOM works at compile-time, we need a different configuration file that tells us where to find the provider. Additionally, the selected provider component needs to get included in the dependencies (i.e., in the .pom file of Maven).

5.1.3. Java Annotations

To generate smart contracts representing user-defined tables, we need metadata. This metadata describes tables and their attributes. Java provides multiple ways to store and gather metadata (e.g., as XML files). In our case, we decided to use Java annotations as most Java database-related frameworks, such as the Java Persistence API (JPA), already use them. Annotations are a form of syntactic metadata added directly to the source code. This allows the user to annotate the files which are already existing. Java annotations start with the @ symbol. An example of such an annotation is in listing 5.4. It shows the User class, which gets marked as a database entity with the @Entity annotation. The @Column annotation defines that the entity has a username column. The result would be a user table with a single column named username.

```
@Entity
public class User {
    @Column
    private String username;
}
```

Listing 5.4: JPA annotation example

5.1.4. Apache Maven

To simplify SmartCOM's usage, we decided to use an open-source build management tool called Maven. It ensures that the build process runs in the correct order and that all dependencies have the right version. SmartCOM consists of separate components split into modules. Modulization allows having different versions of a module (e.g., for backward compatibility) and eases replacing them. Although SmartCOM uses Maven, it would also work with other build tools like Gradle, if configured correctly.

5.1.5. Using a Grammar for Parsing and Mapping

Before deciding on Velocity, we tried to map a grammar file representing Java to another grammar file describing Solidity using ANTLR (a parser generator). While reading some simple classes was relatively easy, it became evident that we would need to either restrict a user to a given style or have a complex structure of grammar files importing sub grammar files (e.g., for method parsing). While this is possible, it still requires some logic written by a developer that selects the recognized production rules and generates a file. As a result, it would simply generate files, that need to follow strict grammar rules. Therefore, one would obtain a similar result as a template engine like Velocity. Another thing to consider is that when converting the POJO back to the input language, we would also require a Velocity file or a grammar file. The steps at compile time would be to parse user class, convert to Solidity contract, and write class capable of converting solidity instances back to user instances. The problem with this is that we can't fill instances of other programming languages. Think of it as an input language Python and Java for SmartCOM. With Java, we can not instantiate Python instances during runtime (within

the Java Virtual Machine). A solution to this problem would be a JSON representation. The reasons for not implementing it this way were, that it would not be a classical ORM approach anymore. A solution for this problem is implementing Representational State Transfer (REST) interface that converts JSON inputs to the objects in the native language (e.g., JSON to Java). The advantage of this approach, is if an object of a Object Oriented Programming Language (OOPL) is convertible from and to a JSON representation, then it fits the framework.

An example would be a Python API that sends JSON representation of Python object. The Java SmartCOM implementation receives the JSON representations and stores it on the blockchain. By hiding the implementation, the user could use it as a classical ORM system. Another reason is that it is harder to implement and maintain than a template framework (especially finding an appropriate set of grammar files). Nevertheless, it would allow multiple input languages without rewriting the code of SmartCOM.

5.2. Solidity Showcase

At first, we start with discussing the Solidity provider implementation showcase. This showcase is an example of a user project, that stores, the data in a database-like manner on the Ethereum blockchain. Each user of the application needs a wallet connected with the blockchain. As each operation costs Ether, users need the necessary funds to execute transactions. These funds can either be predefined or acquired through mining. The single POJOs are represented as Solidity smart contracts.

5.2.1. Required Software

The following showcase requires the following software. Appendix A.1 (startup description) describes the concrete versions and links to the freely available software requirements.

Ganache-cli or Geth

To keep the showcase simple to replicate, we decided to use the Ethereum protocol implementations, that can also be used locally on a single machine for testing. The two main options herby are the development tool Ganache-CLI which is part of the Truffle suite. Truffle is an open-source development environment and testing framework. The other option is one of the three original implementations of the Ethereum protocol called Go Ethereum (Geth). In Ganache, to get the same cryptocurrency wallet addresses whenever testing, a mnemonic is used. In Ethereum, mnemonics are a set of words that work as seeds for the cryptocurrency wallet address generation. In the case of Geth, we use the Genesis file and the key store to guarantee that our cryptocurrency wallet addresses are valid and have a starting balance.

Solidity Compiler

The Solidity compiler (solc) gets used to compile Solidity files (.sol). The results are a binary file (.bin) and an Application Binary Interface file (.abi). The .bin file is simply a binary representation of the compiled code. The .abi file describes the deployed contract and its functions. We need to use the correct compiler version (as not all versions are backward compatible). Additionally, the Solidity compiler location needs to be in the path such that SmartCOM can use it. SmartCOM also uses the ABIEncoderV2 (which is activated by default starting from version 0.8). The pragma for ABIEncoderV2 can be used in older Solidity versions and allows for structs and arrays, making working with multiple entries or class structures much more accessible.

Web3j

Web3j is a library that connects Java applications with the Ethereum blockchain. The main features we use are the Ethereum JSON-RPC client and the auto-generation of Java smart contract wrappers. These wrappers get generated from compiled Solidity code. The wrappers act as a regular Java object which executes transactions on the blockchain when called. Web3j may become part of the Epirus framework in the long term. As we don't use the additional Epirus functionality, we still use standalone Web3j.

5.2.2. Ethereum Code Generation Process

Figure 5.1 shows the code generation process of a user project with annotated POJOs to the deployment on the Ethereum blockchain. In principle, annotated code gets processed and parsed for information. This information is then used to generate Solidity contracts. These contracts then get compiled with the Solidity compiler. Out of the resulting files, Web3j generates Java wrappers. These wrappers include methods to call every function of a smart contract. If this process is successful, Java data repositories get created. Data repositories then use the functions of the Java wrappers known to us due to the smart contract code generation. The user then can use the data repository to deploy or alter smart contracts representing the user POJOs on the Ethereum blockchain.

5.2.3. Solidity Architecture

Firstly, we will discuss the Solidity architecture. The Solidity architecture is the target of the metadata mapping as our Solidity smart contracts are similar to a database structure. In the next subsection, we discuss the Java architecture used for the Solidity provider implementation. Figure 5.2 represents the Solidity blockchain provider architecture. The architecture consists out of the following three main components.

Database Contract

The database contract holds the connection (i.e., contract address) to every table, represented as table contract instance. There exists a table instance for every user-defined

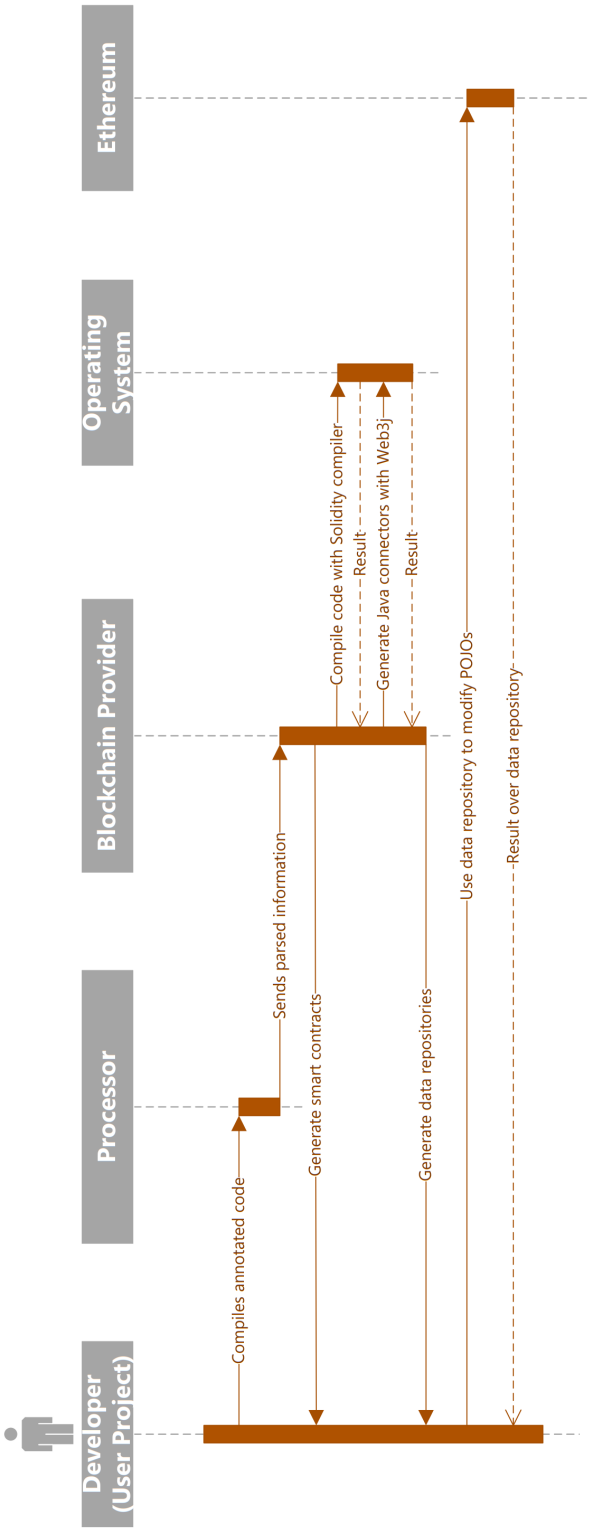

Figure 5.1.: Ethereum code generation sequence diagram

POJO. The table class is needed to add instance relationships by their auto-generated identifier. Storing the identifiers of the entries helps in handling duplicates and updates. Each table gets created upon creating the database. The database contract smart contract address is the only smart contract address, that needs to be persisted by the application to have access to the tables and instances (as the database contract stores all information).

Table Contract

A table gets generated for each user-defined POJO (for example, InstitutionTable for a POJO called institution). The table class is similar to a table in a SQL database. It holds references to instance contracts. When a new instance gets created, a counter gets incremented, and the model receives the unique identifier (autogenerated id). A Solidity table implements a Solidity interface such that there is a guarantee that at least the necessary *get* function exists. The *get* function checks, if related instances exist and get their smart contract addresses. A table also stores a connection to the database. This connection gets used when relations outside of a table are validated.

Instance Contract

The instance contract represents an entry of a table. It may also reference different contracts due to relationships (e.g., one-to-one). Addresses do not need to be stored on the Java side as the table contract stores them on-chain. An instance holds a connection to the table.

Example

Figure 5.3 shows a POJO mapping example with an institution contract having an smart contract address, an *ID*, a version, a hash code, a deleted flag, a name attribute, a table, a single relationship to a user contract (called admin), and multiple connections to the user contract (called users). We require a smart contract address for single relationships, that allows us to find the contract on the blockchain. Additionally, we store the *ID* for faster checking whether a connection exists. As we do not store smart contract addresses off the chain, we require a relationship to the table, which has a relationship to the database. If we need a certain smart contract address, we can query the database contract for the table. Each table then contains a map construct connecting *IDs* of single mapped POJO instances with the number in which they get appended as a relationship to the contract (one for the first connection). The relationship order reference is essential as we need it to get the contract address. In this case, connecting the identifier directly with the contract address is not possible, as we have no feasible possibility to search for filled entries in a mapping. As Solidity does not store the relationship size, we need to keep a count variable up-to-date. By knowing the total number of relationships, we can iterate over the relationship map construct, starting with the first added relationship one up to the map constructs size.

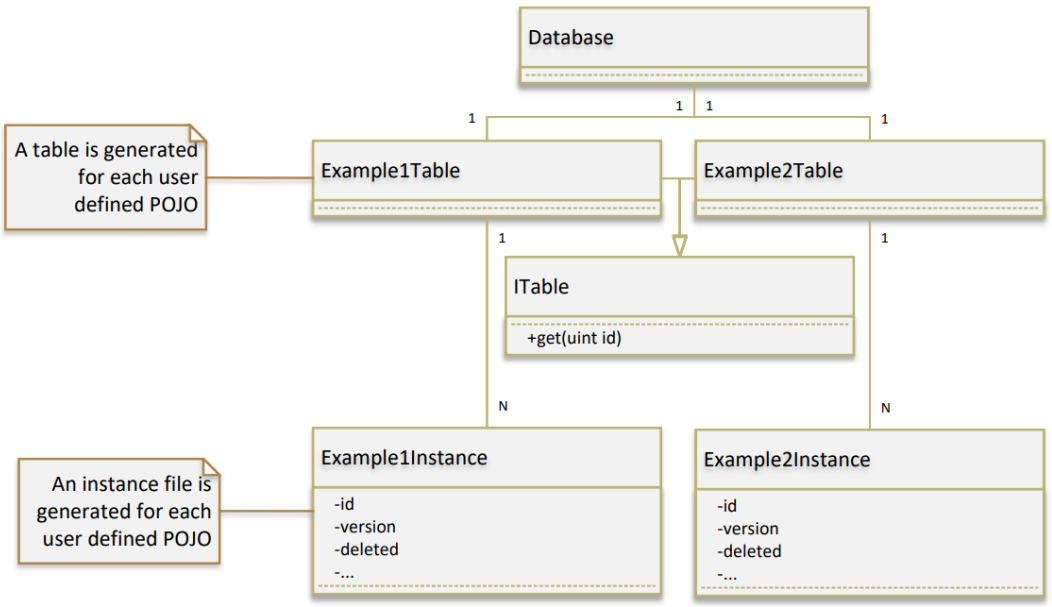

Figure 5.2.: Solidity showcase architecture

Connections between different mapped POJO instances are either single or multiple connections. For multiple connections, we store a mapping, that stores all connected mapped POJO instances in a mapping. Additionally, we use an *ID* map such that checking for existing connections by *ID* is faster. This map is the connection between a POJO *ID* to an *ID* in the mapping, thus giving us the smart contract address holding the information to the POJO. The table contracts hold a connection to the database contract (for the case that the underlying instance needs information about other tables), an instance count, and a map construct that connects an *ID* of an instance to its smart contract address on the blockchain.

Note that the count variables never decrease, as it would be more work resizing a mapping than invalidating entries. By invalidating, we mean that the mapping's entry will get an invalid smart contract address upon removing a connected instance from a multiple mapping (i.e., 0x123 $\rightarrow$ 0x000). Additionally, the *ID* of the removed connections will become 0. As a result, if we load a multiple mapping, SmartCOM checks if the *ID* is either 0 or the contract's smart contract address is an empty smart contract address. With this information, we know that it is no longer present and can be re-added.

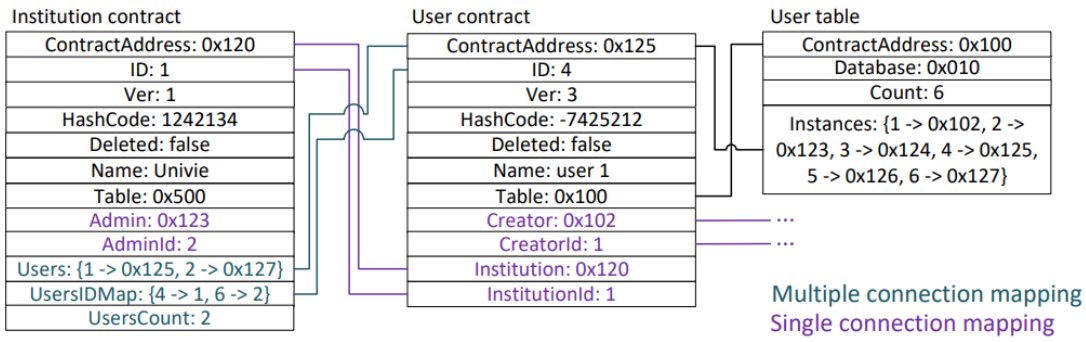

Figure 5.3.: Solidity POJO mapping example

Solidity Details

We used the ABIEncoderV2 as it reduces the Ethereum calls needed to handle arrays or structures. Using Array structures is needed as most of the writing latency comes from waiting for a block (containing the transaction) to be included in the blockchain. Therefore, we reduced Web3j checks for a new block (default 15 seconds) to reduce the waiting time. Additionally, we changed some parameters [Smi17]. Note that if SmartCOM would interact with the main Ethereum network, reduced waiting time would not have much effect, as the wanting time is around 13 seconds (state April 2021 [Eth21]).

5.2.4. Java Architecture

Figure 5.4 shows the Java architecture for the Solidity showcase the architecture consists of four components. Firstly, the table representation, similar to a database table, is a list of entries. Secondly, the repository interface which guarantees a minimal set of methods over the different implementations. Thirdly, the repository which is the interface for an application to interact with the blockchain database. Lastly, the database is the connection to the Solidity contracts.

Example Table

The ExampleTable (e.g., CourseTable.java) is a Java connector for a user POJO called Example. This class is generated out of the ExampleTable.sol file and is a simple auto-generated Java connector from web3j.

Repository Interface

The repository interface interface is shared over all blockchain provider implementations and guarantees a minimal set of methods. These methods include the CRUD methods as

well as trace, and user management methods.

Solidity Repository

Solidity Repository handles the Web3j initialization based on the user configuration file and provides some functions needed by repositories. Note that Web3j does not map Solidity interfaces to Java when converting from Solidity to Java. Therefore, generics are only representable as contract methods (storing value, type as a text, and cast the generic attribute back). Although the tables (e.g., CourseTable.java) need to generate all methods guaranteed by the Solidity contract interface ITable.sol, there is no way for Java to detect the interface as a pattern in Java. To allow persistence SmartCOM loads the database contract instance smart contract address from the config.yaml file either provided by the user or stored from a previous session (i.e., when restarting the application). A new database contract will get generated, if no database contract instance smart contract address is found within the config.yaml.

Example Repository

Example Repository (e.g., CourseRepository.java) defines which table instance gets loaded from the database. If there is no entry in the config file, the repository will generate a new entry and write it back into the file such that it is readable from storage at the start of the application. The repository also implements the Solidity repository, which guarantees some functionality (i.e., CRUD functionality).

Configuration File

A simple YAML file holds the smart contract address of the database, the admin's cryptocurrency wallet address paying for administrative contracts (such as database creation), and the blockchain URL. Here it depends, if the user accounts are unlocked or not. For Ganache, the contracts are unlocked, while Geth needs a special configuration. If the accounts are not unlocked, it is required to give the location of the credentials as well as a password to encrypt them. An example of locked accounts is shown in the listing 5.5. Listing 5.6 shows how it looks in an unlocked situation.

```
1 credentialsLocation: "C:/Users/Martin/Desktop/git/
    masterthesis_martin_pfitscher/prototype/geth/
    blockchain/test/data/keystore/UTC--2021-03-08T18
    -39-57.733649300Z--
    cd1a5d39482dd9aaaa92f3c9f24ff309cb3829db"
2 adminPassword: "123"
3 unlocked: "false"
4 blockchainURL: "http://localhost:8545"
```

Listing 5.5: Locked YAML configuration file

```
databaseAddress: "0
    x15fb3453fcbd087b5a022c50adcc97e718b19cd3"
unlocked: "true"
blockchainURL: "http://localhost:8545"
```

Listing 5.6: Unlocked YAML configuration file

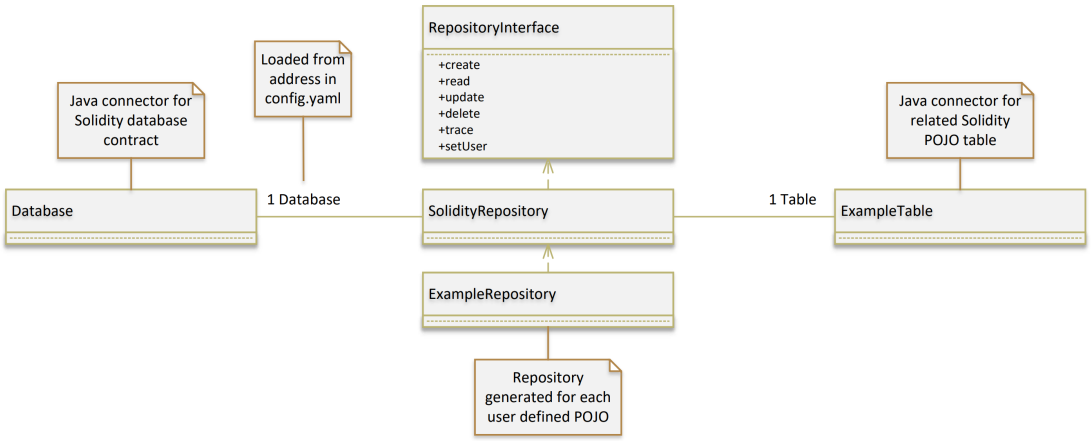

Figure 5.4.: Java architecture for the Solidity showcase

5.2.5. Add Operation Example

Figure 5.5 shows the sequence of operations needed to add a single instance in Ethereum. At first, the user calls the add method of a data repository to add a POJO called *User*. This POJO has a name represented by a text and a date represented by a class called *Date*. In the next step, the POJO attributes get converted. As no date data type exists in Solidity, we need to convert it to a different type. In this case, the translation file defines it as a text. Before we can add an instance, we need to request the next *ID*. After requesting the next *ID*, we can add a new user by calling the add method on the user table. The user will then receive the *ID* as well as a relation to the table. The table relation is needed to add new connections. If relations were present, the user contract would query the connection table by requesting it from the database contract (connected to the user table). The connection table then checks, if the *ID* is valid and returns its address. In this example, we have no connections. Therefore, no additional check is required. The user table then adds the instantiated user contract to a mapping. The user table then stores the reference in a user mapping. By returning a transaction receipt, we can check if the operation was successful and return the eager POJO.

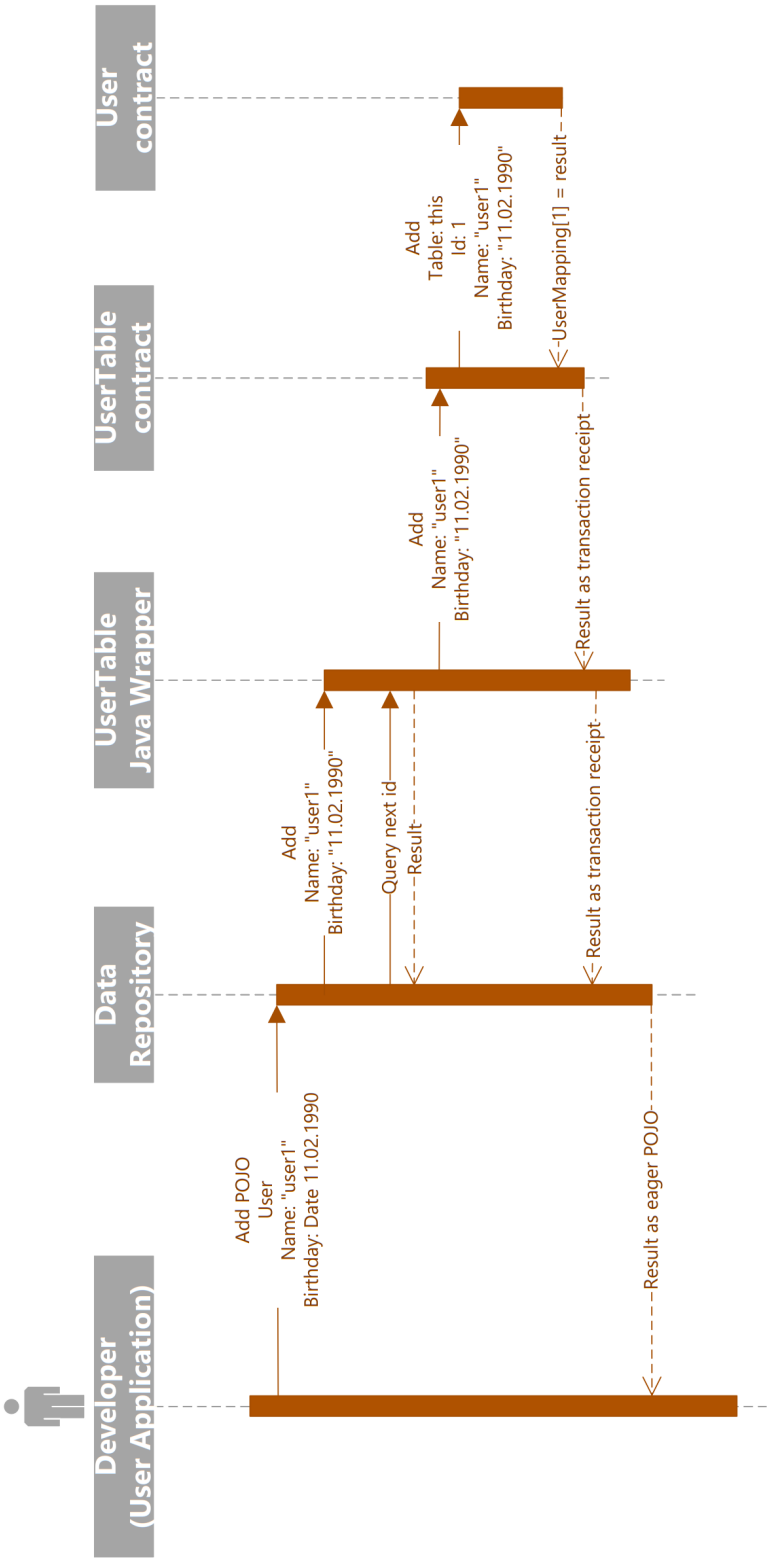

Figure 5.5.: Solidity add operation sequence diagram

5.2.6. Solidity User

Solidity user class enables to changing the user by using the user method in the repositories. Depending on the user configuration, it is possible to change the user with the credentials and a password (when accounts are locked) or just with the public cryptocurrency wallet address (when unlocked).

5.2.7. Solidity – Java Conversion

Some conversion needs to happen to generate methods that use the user-defined POJOs instead of the smart contract instances. To translate the types of Java to Solidity, we use a target translation file. This file is extendible if new types need to be included. Additionally, the original types get recorded to convert the Solidity contracts back to Java POJOs. If a contract gets loaded from the blockchain, the contract and its relations get converted to a Java POJO. All loaded connections get stored in a hash map with the hash code (including all attributes and the version) and the loaded object (the reference to it). This information gets used to prevent a cycle when loading relations. By doing so, already loaded objects are not reloaded.

To trace stored entities, not only the *IDs* of the related mapped POJOs are required, but also their version. To get all versions of a given corresponding instance, SmartCOM calls the trace method also on them. The exact instance and all related instances at the time of change get loaded.

5.2.8. Tracing in Solidity

We used Solidity events to trace changes. These events belong to a smart contract address, which is, in our case, a unique identifier for a POJO. We retrieve these events by using Web3j scanning methods. As writing in the blockchain can become very expensive, we restricted the event content to the attributes of the instance and the connections in the form of *ID.Version* (with a dot as separation) and multiple *IDs* as CSV list of the form *ID.Version*. Figure 5.6 shows a trace operation of a single unconnected instance. At first, a user application calls the trace operation of the data repository with a given identifier of a POJO. As events are traceable by contract address, we first need to find the address of a given POJO with the identifier. After finding the address, we can scan the blockchain for events emitted by the contract. After getting the events, we check them for connections. If there are connections, we also need to call a trace operation on them. But in our example, we have an unconnected instance. Therefore, we can convert the events to trace events. Trace events contain an eager user POJO, a timestamp, and a text identifying the sender. The user then receives the results as a list. Figure 5.7 shows the result of a trace operation where we search for the institution with the *ID* 1. The events emitted by the contract of institution 1 contain the *IDs*, version, name, a deleted flag, all attributes (e.g., name), the sender, and the connection *IDs* (e.g., user *IDs*). For the users trace query with *ID* 1 and version 1, we have to consult the traces of the user contract with *ID* 1. The combination of the trace results will give a complete trace of the POJO at a given

time.

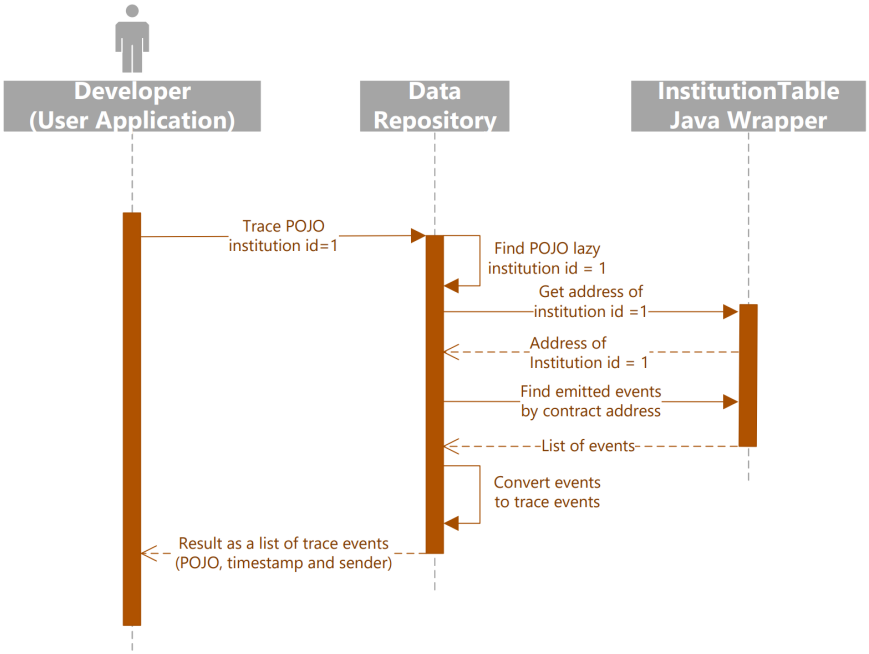

Figure 5.6.: Example Solidity trace sequence diagram

5.2.9. Create/Update on Relations Policy

If a related instance is unknown to the blockchain, then the corresponding instance gets created. If the version of a related instance is the same as the instance with the same *ID* on the blockchain, but the hash code varies from the stored instance, then a related instance will get updated. If relations changed or if they are new, then they get updated or persisted. In all other cases, related instances don't get updated or persisted.

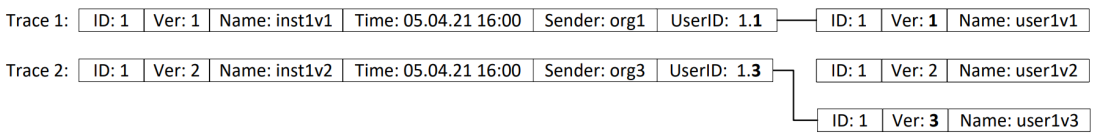

Figure 5.7.: Example Solidity trace result

5.3. Hyperledger Showcase

In this section, we are discussing the Hyperledger provider implementation showcase. This showcase is an example of a user project that stores the data in a database-like manner on the Hyperledger Fabric blockchain. To simplify the implementation, we chose to use Java chaincode. In Hyperledger Fabric a user needs to be connected to a node. An example would be the accountant of a company that needs access to the application to determine a company's taxes.

5.3.1. Required Software

In this section, we will discuss the necessary software for the Hyperledger showcase.

Docker

Docker is software to isolate applications into containers. These containers can run different operating systems and be configured in various manners. In the case of Hyperledger Fabric, chaincode and the blockchain is run in a containerized manner to allow more smart contract languages, constructs, and types than most smart contract-based blockchain systems. Typically, certain programming constructs, such as construct based on randomization, are not allowed in a smart contract language. This is normally done as all nodes need to have the same state. Nevertheless, by running the chaincode in containers, more constructs can be allowed as the container can be stopped/restarted if it does not respond or provides a different result.

Hyperleger Fabric Binaries and Docker Images

Currently, only an install script for Hyperledger Fabric exists. It installs all binaries needed as well as downloads all docker images. These docker images then get started with a starting script. To test Hyperledger Fabric, it is best to use their provided test network and their starting scripts.

Gradle

Gradle is used to compile the chaincode. Gradle is a build management tool similar to Maven. It gets used as the Hyperledger Fabric network requires it, although it would be possible to rewrite it to Maven.

5.3.2. Hyperledger Code Generation Process

Figure 5.8 shows the code generation process of a user project with annotated POJOs to the deployment on the Hyperledger Fabric blockchain. In principle, annotated code gets processed and parsed for information. Out of this information, a chaincode project gets generated. These contracts then get compiled. If this process is successful, Java data repositories get created. Data repositories then assume that the transactions defined in

the chaincode are accessible over a gateway. In our test cases, the script which starts the Hyperledger Fabric test network also deploys the chaincode project. The deployment of the chaincode could also be moved to the blockchain provider if needed. The user then can use the data repository to deploy or alter the chaincode managing the user POJOs on the Hyperledger Fabric blockchain.

5.3.3. Architecture

This section will discuss both the Hyperledger Fabric chaincode and the Java architecture used for the Solidity provider implementation. In Hyperledger Fabric, the chaincode gets stored in a separate project that can run within a Docker container.

Chaincode project structure

The chaincode project is a Gradle-based project. We kept Gradle as a build tool for the chaincode as all Hyperledger samples used Gradle as well, and it would require rewriting parts of the network initialization to use Maven or other build tools. The project structure gets shown in Figure 5.9. The Gradle files get copied from the blockchain provider implementation project to the new chaincode project. Then blockchain provider will copy the counter file as the operations class needs it. The user-defined POJOs will get generated into the generated folder (e.g., UserOperations). The chaincode project is not generated into the user projects target folder. This is the case as Gradle, and Hyperledger Fabric requires us to copy Gradlew wrappers to the project, which will under some operating systems (such as Windows) cause problems with clearing the target folder.

Chaincode Architecture

Hyperledger stores data in a key-value map consisting of strings. Figure 5.10 shows the architecture of the chaincode. Therefore, we can have a key naming convention instead of a table class. Instances get named with the class instance name and their *ID* (e.g., user instance one would have key user1). As we don't know how many instances exist by this naming convention, we need a counter. This counter gets stored with the key named after the instance class name and counter (e.g., userCounter). Transactions need to be defined to interact with the stored instance from the outside. These transactions get defined in the operations files (e.g., UserOperations).

Counter

The counter stores the number of instances of a converted POJO currently exist in the blockchain. It gets used as an autoincremented value for the identifiers of the instances.

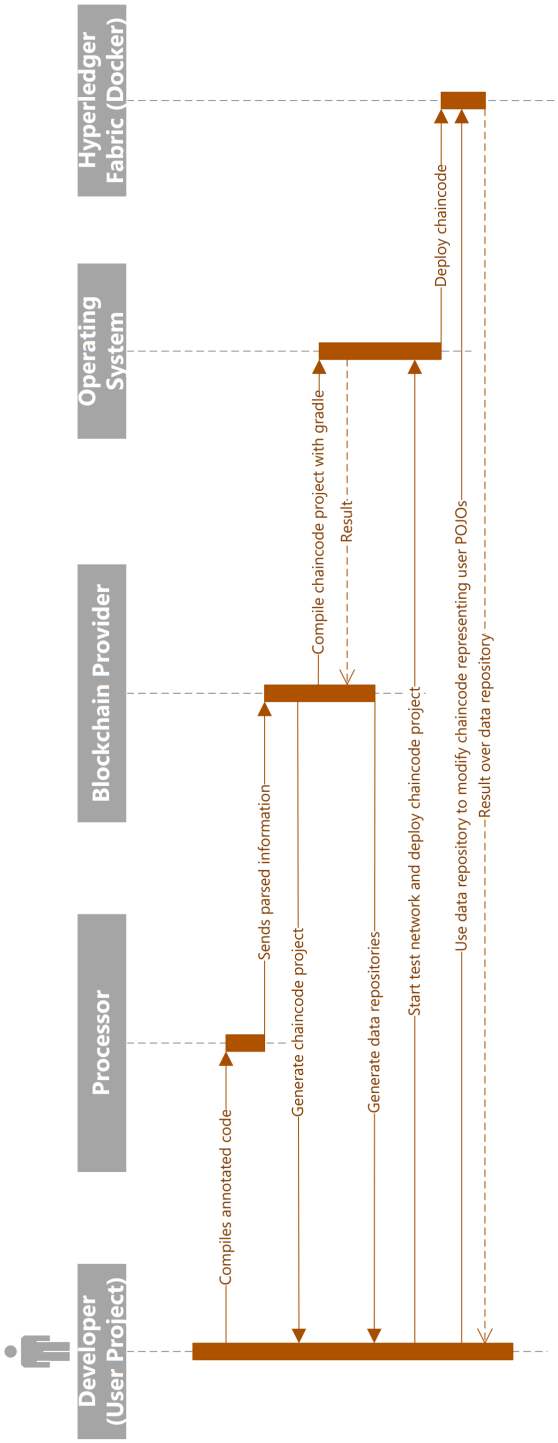

Figure 5.8.: Hyperledger Fabric code generation sequence diagram

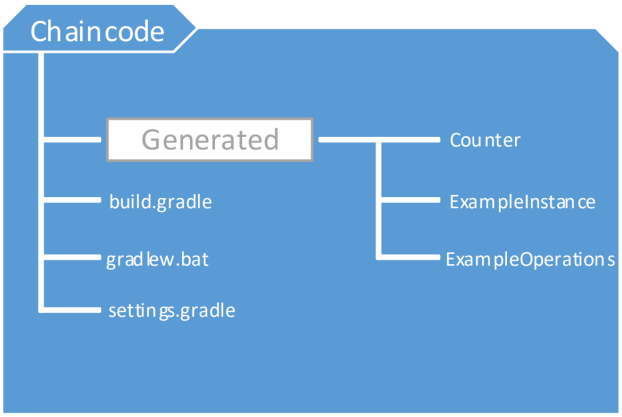

Figure 5.9.: Chaincode project structure

ExampleInstance (e.g., User)

An instance is a generated class from the user POJO and stores all annotated attributes of the POJO and the id, the version number, and a deleted flag.

ExampleOperations (e.g., UserOperations)

An operations class will be generated for each POJO and will contain all transactions regarding it. Most of the storage logic is within these transactions. Therefore the generation of CRUD methods is also defined in the operations class. The Java API can then call these transactions.

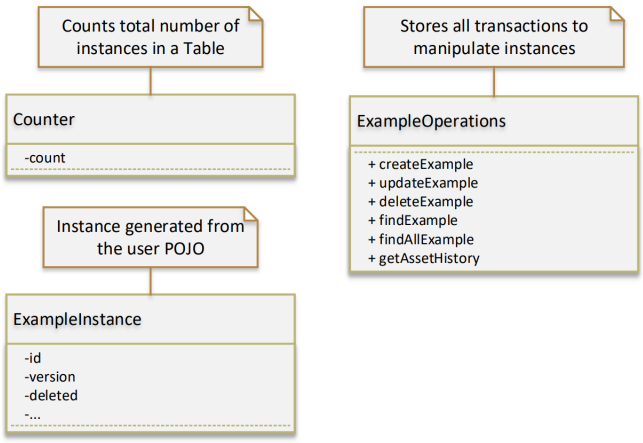

Figure 5.10.: Chaincode architecture

5.3.4. Add Operation

Figure 5.11 shows the sequence of operations needed to add a single instance in Hyperledger Fabric. Although this diagram is for the add operation, it is very similar to the find, delete and update methods. At first, the user calls the add method of a data repository to add a POJO called *User*. This POJO has a name represented by a text and a date represented by a class called date. In the next step, the POJO gets converted to textual attributes as the interface given by Hyperledger Fabric API does only accept text as input. The textual form of the POJO then gets sent to the operations chaincode, which defines all transactions (also delete, update, read and trace). As Hyperledger Fabric works as key-value storage, we decided to have a key-value pair for every instance. We decided to have a counter instance for every POJO type to know the number of storage objects. When creating the first instance, we need to create the counter as well. For every additional instance, we increment and get an identifier from the counter. The counter also returns the total number of instances in the case of a find all. After getting the *ID* of the new instance, we store the chaincode representation of the POJO with the key *UserId*. The instance chaincode is generated depending on the translation file and the VTL file. In this case, we store the date as text. In the case of an update or delete, we can find the instance by retrieving the key *User + passed ID*. After persisting an instance, we return the result in a serialized manner. The data repository converts it then back to an eager POJO for the user.

5.3.5. Trace Operation

Figure 5.12 shows how a trace operation in Hyperledger Fabric works. At first, a user application calls the trace operation of the data repository with an identifier. The data repository then delegates the task to the operations chain code. The evaluation chain code then gets the historic states for the key "Institution1". These results get converted to trace entries and returned to the data repository, which then uses typecasting to return the trace events as a list.

5.3.6. Java Architecture

Figure 5.13 describes the SmartCOM Hyperledger Fabric Java architecture. As our chaincode is Java, there is no need for special Java connectors. The generated example repositories (e.g., UserRepository) contain all implemented methods of the repository interface. Additionally, in Hyperledger, we need to register users with the application. This means only administrators predefined in Hyperledger Fabric can be added as admins to our application. As soon as an administrator gets connected to our application, a user can be created with rights to use the application by adding them as users such that they can commit actions under the name of the administrator. Administrators and users can get added by using the user management methods. To add an administrator the username, password, host, certificate authority, the membership service provider, and the location of the .pem file is needed. This information is the same that is given when creating the

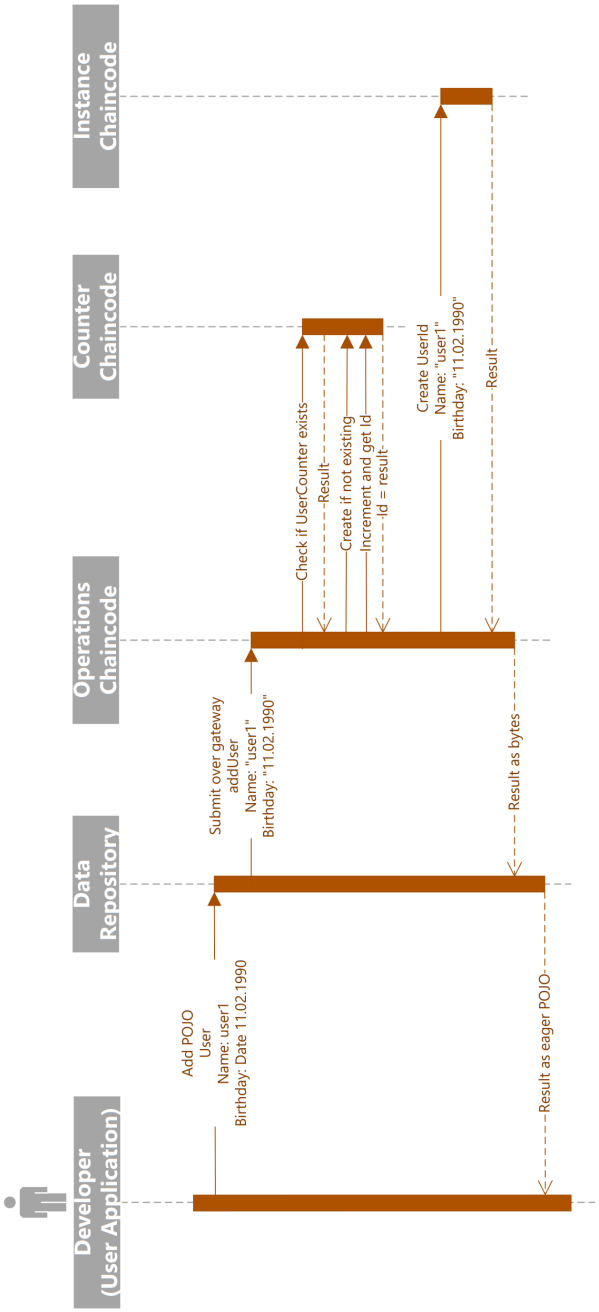

Figure 5.11.: Hyperledger Fabric add operation sequence diagram

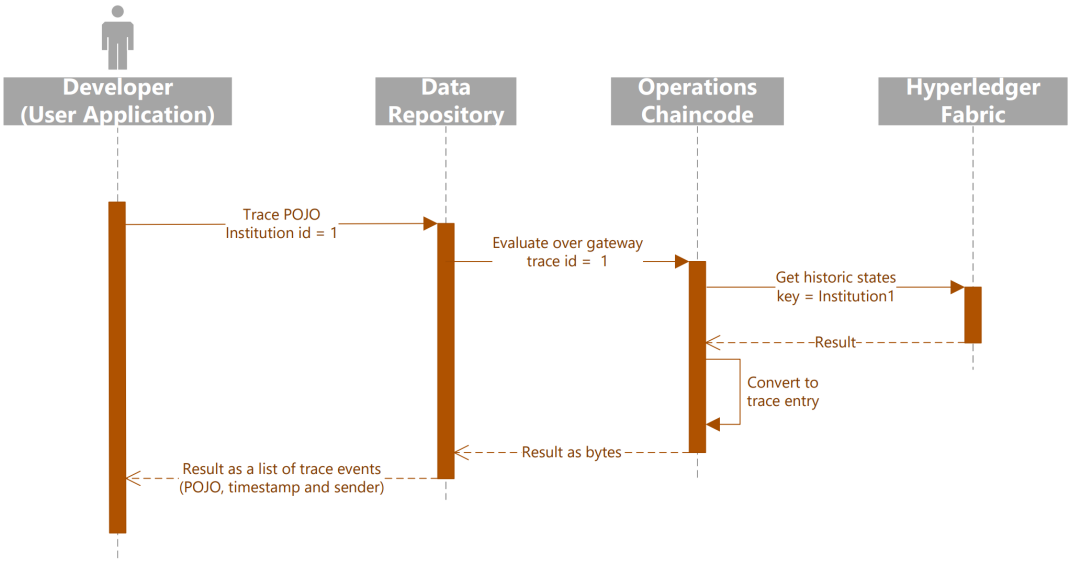

Figure 5.12.: Hyperledger Fabric trace operation sequence diagram

user on the Hyperledger Fabric network. Once an administrator or a user gets created, it can get used in the application by using the set user method in any repository and passing a Hyperledger user object (with the username as an identifier and the network configuration path where the user was defined).

5.3.7. Configuration

Listing 5.7 shows a configuration file for Hyperledger Fabric. In the case of Hyperledger, the configuration contains the host, admin username, admin password, blockchain URL, peer organization location, wallet location, and membership service provider. This information is needed to create the first admin in our application. Additional users that belong to the same organization can get added with less information using the Hyperledger user method to create users. The same information as in the configuration is needed to create another admin user. The information can be passed to the Hyperledger user method create admin. This is necessary as the different admins can be part of different organizations with a different blockchain URL and may be stored in different locations.

```
1 blockchainURL: "http://localhost:7054"
2 adminUsername: "admin"
3 adminPassword: "adminpw"
4 peerOrganisationLocation: "../test-network/organizations
    /peerOrganizations/"
5 walletLocation: "wallet"
```

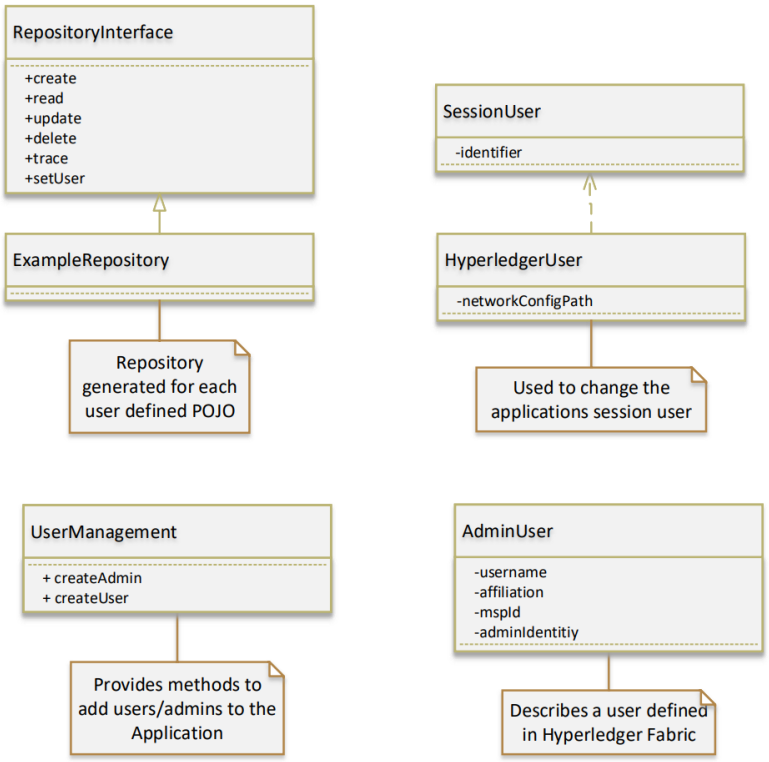

Figure 5.13.: Hyperledger Fabric Java architecture

```
6 mspId: "Org1MSP"
```

Listing 5.7: Hyperledger YAML configuration file

5.3.8. Hyperledger Fabric Chaincode – Java Conversion

As our chaincode is also Java-based, only the annotations have to get converted. All data types will just get copied as they were. Nevertheless, only annotated attributes will get converted. In Hyperledger Fabric, in addition to the instances, we store every relation and its contents. A history entry gets created that is our trace of an instance with a given version. As a relationships in Hyperledger Fabric are not get defined by a certain smart contract address or reference, Hyperledger Fabric does not guarantee to get relationships to other chaincodes up to date from the blockchain (as Hyperledger Fabric does not know about the relationship). To get the newest version, we check if changes occurred at every load/write operation. Although this is computation-intensive, it requires only loads that a single node can handle in the network.

5.4. Evaluation

This section contains the evaluation of SmartCOM. Automated tests evaluate the system. In the thesis, we will only discuss performance-related tests. The tests were executed time independently on the same machine (CPU: AMD Ryzen 3700X, RAM: 16 GB 3200MHz, GPU: Geforce RTX 2070 Super, SSD: 1.5TB NVME SSD, OS: Windows). The main goals of these tests were to evaluate the system, find performance bottlenecks, and determine scalability problems. We compare the Ethereum based implementation running on Geth with the Hyperledger Fabric provider implementation for the test. We decided not to run our Ganache tests as it is only a testing environment (therefore not viable in production).

5.4.1. Models

Figure 5.14 shows a class diagram of the test models. Each user-defined model needs to extend the abstract model as we need an id, version, and a deleted flag. We selected an education test scenario with users and institutions. Each institution has a name, users, and an admin. We refer to the institution class as a class with multiple connections in the tests as it validates how multiple connections (*users* field) affect the performance. Users have a name, some European Credit Transfer System (ECTS) points, a birthday, a creator, and belongs to at most one institution. We refer to users as the single connection class as it validates how single connections affect the performance. To validate the getter/setter selection based on the modifiers, we included some public attributes. Additionally, we used two attributes (ECTS and birthday) which require a translation to other types in Solidity.

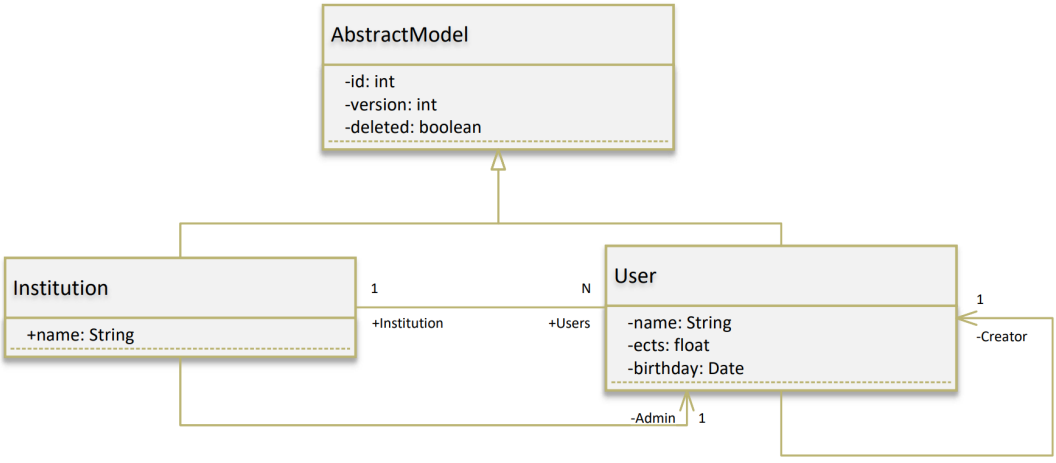

Figure 5.14.: Test model class diagram

5.4.2. Blockchain Configuration

In this section, we are discussing the blockchain configuration for the tests.

Ethereum Configuration

The Ethereum network was run locally on Geth with the parameters listed in listing 5.8. The HTTP parameter enables the HTTP-Remote Procedure Call (RPC) server, and the HTTP address is localhost. To be able to add additional nodes, we set the network *ID* and a port. SmartCOM requires us to set the HTTP port as defined in the configuration file. Additionally, we need a data directory for the node, select the API which should be exposed by the RPC server, and enable all domains. At last, we use the default settings (i.e., fast sync mode and a cache of 4096 megabytes).

```
1 geth --http --http.addr 0.0.0.0 --networkid 1234 --port 30303 --http.port
      8545 --datadir data --http.api="eth,net,web3,personal" --http.
     corsdomain "*" --syncmode fast --cache=4096
2 geth --datadir data_node1 --networkid 1234 --port 30301 --rpcport 8101 --
     ipcdisable
```

Listing 5.8: Geth parameters

The executing account was prefilled with some Ether (Ehtereum cryptocurrency) and will be rewarded with new Ether due to the mining process. For our tests, we used a two-node environment with enabled peer searching.

Hyperledger Configuration

For Hyperledger, we used the test network from their documentation with two organizations. The chaincode was executed on Docker. To run the test-network [Doc21] on Windows, some minor changes to the directory paths had to be made (to the shell scripts). Docker was configured on the test machine to use Secure Virtual Machine (AMD-SVM) instead of Hyper-V.

5.4.3. REST Demo Project

In this section the REST demo project is discussed. At first we discuss, the architecture and then the usage of the system as well as the results to expect.

Architecture

The REST demo project is a simple Spring-based Maven project. It provides the ability to interact with the model of Figure 5.14. To reduce the number of calls made to the blockchain, it accepts only JSON representations of user POJOs. The reason behind this is that SmartCOM is optimized to handle POJOs directly and needs to load objects if given an id. If we have a full JSON POJO we can give this information to the blockchain provider, which can save it without further changes. In the case information is not valid or outdated, it will simply pick the newest version. Figure 5.15 shows how the REST demo

project interacts SmartCOM. A user can send POST, PUT, GET and DELETE requests over HTTP to the server which is hosting the REST demo. The REST demo project will the repositories generated for the demo project. The demo project itself generates these repositories by using SmartCOM. The blockchain providers within SmartCOM then communicate with the selected blockchain.

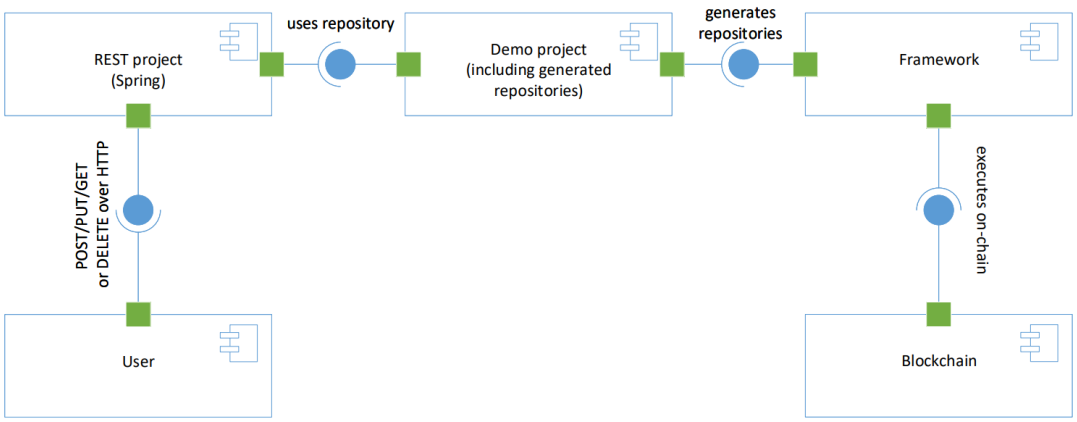

Figure 5.15.: Architecture of the REST demo

Usage

In order to run the REST demo project Maven is needed. To start the project run at the location of the .pom file in the rest demo project *mvn spring:boot-run*. Now it will start and run on localhost if port 9170 is free (otherwise alter the port in application.yml under src/resources). The easiest option to test the REST demo project is by using Postman. Once given access to the repository, a Postman file is ready to use with all supported queries and example parameters. The concrete REST endpoints (for the *user* POJO) are:

Add user. A user is added by sending the JSON representation via POST to SERVER:PORT/user. The result is the JSON representation of the added POJO.

Get user. Returns a user if given an *ID*. The user with *ID* one that should only be returned if not deleted is retrieved by sending GET to SERVER:PORT/user/id=1&deleted=false. The result is the JSON representation of the POJO.

Update user. A user is updated by sending the JSON representation via PUT to SERVER:PORT/user. The result is a JSON representation of the updated POJO or in the case of a version conflict the same POJO.

Delete user. Deletes a user when given an *ID*. The user with *ID* 1 is deleted by sending DELETE to SERVER:PORT/user/id=1.

Getting all users. All users are retrieved by sending a GET request to SERVER:PORT/users. The result is the JSON representation of a list of POJOs in JSON form.

Tracing a user. Traces a user when given an *ID*. The User with *ID* 1 is traced by sending a GET request to localhost:9170/trace/user?id=1. The result is a list of trace events in the form of a JSON.

5.4.4. Performance Tests

In this section, we are discussing concrete tests and their results. At first, we always describe the test content and then analyze the results. In addition to the test results table, we have a diagram showing all execution measurements for a test in the appendix A.2.

Creation Tests with Single/Multiple Empty Connections

This test evaluates how long the creation of a simple object without connections to other objects takes. It also indicates if time increases with more instances belonging to the same table. We used the user/institution class for these tests and left all single/multiple connections empty. The test machine executed this test 500 iterations. Each iteration, the test created a new instance. This test is mainly used to test the table structure of the Ethereum provider implementation. Table 5.1 shows statistical metrics, and Figure A.1 shows the performance of each iteration on the single empty connections test. Table 5.2 shows statistical metrics, and Figure A.2 shows the performance of each iteration on the multiple empty connections test.

Analysis The size and methods used do not change between the different instances. As the size single instances do not change, it explains why the time consumption is mostly the same. Having multiple empty connections seems to be slightly more expensive than a single empty connection in Ethereum. Multiple connections take longer due to the larger contract size, mainly due to storage reserved for mappings and the more complex mapping handling methods. It has to be kept in mind that the single connection has two more text attributes. In Ethereum time needed to add new instances to the table also slightly increase in both tests (visible in the trend line). Hyperledger, on the other hand, has no outliers (except for the first entry) as it stores the data in key-value texts. The first entry may be slower due to caching. As a result, Hyperledger is only slightly faster than Ethereum in terms of the average time and median. However, Hyperledger is much more reliable, which can be seen by the straight line in both figures, the minimum and maximum time.

Loading Tests with Single/Multiple Empty Connections

Tests how long eager loading of instances with single/multiple connections takes. In this case, we do not consider semi-eager loading as it would result in the same operation. We

	ETH	Hyperledger
Total time	1066573	1031390
Mean time	2132,69	2062,78
Median	2157	2059
Max time	4842	3621
Min time	618	2053

Table 5.1.: Single empty connections creation metrics. Times in milliseconds.

	ETH	Hyperledger
Total time	1163477	1029406
Mean time	2326,57	2058,81
Median	2280,5	2058
Max time	9092	2325
Min time	150	2051

Table 5.2.: Multiple empty connections creation metrics. Times in milliseconds.

	ETH	Hyperledger
Total time	38569	5096
Mean time	77,13	10,192
Median	44	10
Max time	150	230
Min time	37	8

Table 5.3.: Single empty connections loading test metrics. Times in milliseconds.

	ETH	Hyperledger
Total time	41146	4970
Mean time	82,29	9,94
Median	43	8
Max time	238	223
Min time	35	8

Table 5.4.: Multiple empty connections loading test metrics. Times in milliseconds.

do not consider the time needed to create the instances as we load those generated by the first two tests. The test machine executed this test with 500 iterations.

Analysis Single and multiple mappings did not show a significant difference in load performance visible in Table 5.3 and Table 5.4. In Ethereum, we see a tendency that loading takes either less than 50 ms or over 100 ms. This result persisted over multiple runs and is probably due to caching. The Ethereum trend line (visible in Figure A.3 and A.4) in these tests indicates that the more data we load, the higher is the probability that the instances are not in the cache.

Updating Single Instance Test

This test evaluates how long updates on simple existing instances take. A user instance represents a simple instance. We do not consider the time needed to create the instances as we load those generated by the first two tests. We executed the test with 500 iterations.

Analysis We can see that creation is cheaper for Ethereum than an update operation (visible in the difference between Table 5.1 and 5.5). It is mainly cheaper due to more checks needed to apply when it comes to updating instances. For Hyperledger, an update operation costs the same as the create operation. The reason for this is the way updating works. In Hyperledger, we overwrite the value in the key-value storage. Overwrite means

	ETH	Hyperledger
Total time	1334782	1025733
Mean time	2669,56	2051,466
Median	2619,5	2051
Max time	10968	2081
Min time	114	2045

Table 5.5.: Single instance update test metrics. Times in milliseconds.

	ETH	Hyperledger
Total time	1773631	1025445
Mean time	3547,26	2050,89
Median	3293	2050
Max time	17332	2279
Min time	92	2045

Table 5.6.: Single instance delete test metrics. Times in milliseconds.

we do the same as in a create operation (except for storage reservation). Figure A.5 indicates that Ethereum costs are increasing after updating more instances (around 8% difference between first 250 and last). The reason for this might be caching and a growing blockchain.

Deleting Single Instance Test

This test evaluates how long deleting simple existing instances takes. A user instance represents a simple instance. We do not consider the time needed to create the instances as we load those generated by the first two tests. We executed the test with 500 iterations.

Analysis We can see that creation is cheaper for Ethereum than a delete operation (visible in the difference between Table 5.1 and 5.6). It is mainly cheaper due to more checks needed to apply when it comes to deleting an instance. For Hyperledger, a delete operation costs the same as the create operation. The reason for this is the way deleting works. In Hyperledger, we overwrite the value in the key-value storage. Overwrite means we do the same as in a create operation (except for storage reservation). Figure A.6 indicates that Ethereum costs are increasing after deleting more instances (around 7% difference between first 250 and last). The reason for this might be caching and a growing blockchain.

Trace Single Instance Test

This test evaluates how long tracing a simple existing instance takes. A user instance represents a simple instance. Each instance has three traces from the previous tests (from

	ETH	Hyperledger
Total time	111787	4751
Mean time	223	9,5
Median	250,5	8
Max time	445	250
Min time	139	6

Table 5.7.: Single instance trace test metrics. Times in milliseconds.

	ETH	Hyperledger
Total time	2251931	2061161
Mean time	4503,86	4122,32
Median	4520,5	4121
Max time	8940	4350
Min time	1366	4111

Table 5.8.: Single connection creation test metrics. Times in milliseconds.

	ETH	Hyperledger
Total time	2546535	2061539
Mean time	5093	4123,08
Median	4787	4122
Max time	15614	4147
Min time	1415	4114

Table 5.9.: Multiple connection creation test metrics. Times in milliseconds.

create, update and delete). We do not consider the time needed to create the traces. We executed the test with 500 iterations.

Analysis The Table 5.7 shows that Hyperledger tracing is much faster than Ethereum. Figure A.7 also indicates that the runtimes are much more reliable. The initial spike is probably due to caching. The trend line of Ethereum suggests that the operation is not affected by the table/blockchain size. In comparison to Table 5.3 we see that tracing in Ethereum takes roughly three times the load time for three trace entries. This means an event is not significantly more expensive or cheaper in a local Geth environment. For Hyperledger, on the other side, we see much better scalability where it is actually, on average faster than a single load.

Single/Multiple Connection Creation Tests with a Single Instance as Connection

This test evaluates how long adding an instance with a single connection added to an new relationship takes (e.g., 10 iterations mean that the instance has 10 connections to other instances). We executed the test with 500 iterations.

Analysis Table 5.8 shows how single connections affect Hyperledger and Ethereum. We can see in comparison 5.1 that it takes roughly the same time as creating two user instances. Figure A.9 suggests that both systems are to some extend scalable in this regard. Table 5.9 and FigureA.9 suggest filling multiple connections with one instance much more expensive for Ethereum.

	ETH	Hyperledger
Total time	233465	246726
Mean time	2334	2467,26
Median	2161,5	2463
Max time	5849	2875
Min time	646	2075

Table 5.10.: Adding one connection per iteration test metrics. Times in milliseconds.

Creating One Instance with the Number of Iterations Connections

This test evaluates how long adding an instance with an instance every iteration with the number of iteration connections takes. We used the institution class as an example. All connections got previously created. We executed the test with 100 iterations.

Analysis The Table 5.10 in comparison to 5.2 indicates that creating an instance with multiple connections does not cost significantly more than a single instance with no connections. While Ethereum seems to be getting faster, we validated numerous runs that it highly depends on when the transaction is included in a block (in this case, at the beginning slower than the end). Hyperledger, on the other hand, is getting slower as we add more connections as the text that it needs to store is getting larger. The reason for this is that we keep a textual snapshot of the instance with all connections. As a result, tracing works without consolidating different the history of different objects.

Eager and Semi-Eager Load Tests of Instances with Number of Iterations Connections

These tests evaluate how long the eager and semi-eager loading of an instance with the number of iteration connections takes. We used the institution class as an example. We executed the test with 100 iterations.

Analysis The Table 5.11 in comparison 5.4 indicates that semi-eager loading takes just a little longer than loading an instance with no connections. In Hyperledger, we only read the *IDs* from the connections and fill the eager instance with empty connected objects (except for the id). Similar to Figure A.3 and A.4 in A.11 we see that loading either takes around 50 milliseconds or over 100 milliseconds. Eager load in Hyperledger is simply not checking for updates in the connections. Historical states mean the resulting entry is getting larger with each iteration, and therefore loading and sending are getting slower over multiple iterations. Nevertheless, in the historical Hyperledger states, connections are still up to date as every update, create, or delete operation overwrites it. These improvements are needed, as we see in the Table 5.12 and FigureA.12. The time needed by Hyperledger grows by an average of around 2% with each connected instance as each instance has to be loaded to be up-to-date. On the other side, Ethereum shows a linear

	ETH	Hyperledger
Total time	8726	1343
Mean time	87,26	13,43
Median	49	10
Max time	239	243
Min time	41	6

Table 5.11.: Semi-eager load test metrics. Times in milliseconds.

	ETH	Hyperledger
Total time	431643	4571
Mean time	4316,43	45,71
Median	4153,5	43,5
Max time	12011	215
Min time	96	9

Table 5.12.: Eager load test metrics. Times in milliseconds.

	ETH	Hyperledger
Total time	190235	244420
Mean time	1902,35	2444,2
Median	1981,5	2450,5
Max time	4345	2826
Min time	573	2068

Table 5.13.: Updating an instance with number of iterations connections test metrics. Times in milliseconds.

time consumption where each iteration increases by around 13%. Figure A.12 shows that for Ethereum, an eager load is not feasible. The reason for this is that there are multiple calls needed to get the needed information. A solution to this problem would be to create a function in Ethereum that retrieves all information with one call. One problem with this approach would be that the result size would grow rapidly. Although splitting the result into small chunks would be an option, not all blockchains support it (i.e., Web3j does not support it).

Updating an Instance with a Number of Iteration Connections

This test evaluates how long updating an instance with the number of iteration connections takes. We used the institution class as an example. We executed the test with 100 iterations.

Analysis The Table 5.13 in comparison to 5.10 indicates that updating takes roughly the same time as creation in the case of Hyperledger. This is due to Hyperledger overwriting the value in the key-value storage. For Ethereum, updating is faster than creation. Figure A.13 indicates that Hyperledger takes on average 3% more time per connection for an update due to the larger textual representation and the need to read the updated connections.

	ETH	Hyperledger
Total time	222555	205312
Mean time	2225	2053,12
Median	2118,5	2051
Max time	7127	2067
Min time	181	2045

Table 5.14.: Removing all connections from an instance with number of iterations connections test metrics. Times in milliseconds.

	ETH	Hyperledger
Total time	262555	244595
Mean time	2625,55	2445,95
Median	2371	2437,5
Max time	7901	2890
Min time	693	2068

Table 5.15.: Adding one instance via update every iteration to the multiple connection test metrics. Times in milliseconds.

Updating an Instance with a Number of Iteration Connections

This test evaluates how long removing all connections of an instance with the number of iteration connections takes. We used the institution class as an example. We executed the test with 100 iterations.

Analysis The Table 5.14 in comparison to 5.13 indicates that removing instances is more expensive for Ethereum. The reason for this is that we need to invalidate the connections on-chain. Hyperledger, on the other hand, gets faster as the resulting string is smaller (comparable to the Table 5.2). Figure A.14 indicates that the runtime remains stable with more connections.

Updating an Instance by Connecting It to the Number of Iteration Connections

This test evaluates how long adding the number of iteration connections by an update takes. We used the institution class as an example. We executed the test with 100 iterations.

Analysis The Table 5.15 in comparison to 5.10 indicates that updating is more expensive than creating in the case of Ethereum. This is mainly due to the need for more checks when updating an instance with connections. In the case of Hyperledger, it is the same as the textual representation (i.e., has the same size). Figure A.14 indicates that both provider implementations take longer with more connections.

	ETH	Hyperledger
Total time	1475895	2384
Mean time	14758	23,84
Median	11944,5	25
Max time	45412	225
Min time	592	9

Table 5.16.: Tracing an instance with an increased amount of connections every iteration test metrics. Times in milliseconds.

Tracing Instance with a Number of Iteration Connections

This test evaluates how tracing an instance number of iteration connections takes. We used the institution class as an example. We executed the test with 100 iterations. Each iteration, we load four traces (create, update, remove all connections, add connections via update). Figure A.16 represents the time consumption on a logarithmic scale with the base 10. This is needed as the two approaches perform very differently.

Analysis The Table 5.16 in comparison to 5.7 indicates that tracing in Ethereum takes at least *time to load a single trace* × *number of traces* × *number of connections* milliseconds (shown as grey dots in Figure A.16). As this can take a long time, it is only feasible as a log or overnight task when used on instances with multiple connections. Hyperledger takes for the first 50 instances on average 61% less time to load the trace than for the last 50. But as Hyperledger takes on average 24 ms, it is neglectable.

Tracing an Instance with an Increasing Amount of Traces

This test evaluates how long tracing an unconnected instance with a growing amount of traces each iteration takes. We used the institution class as an example. We executed the test with 100 iterations.

Analysis The Table 5.17 in comparison to 5.7 indicates that tracing in Ethereum takes at least *time to load a single trace* × *number of traces* (visible in A.17). Hyperledger, on the other side, takes for the first 50 instances on average 59% less time to load the trace than for the last 50. But as Hyperledger takes on average 15 ms, it is neglectable.

5.5. Comparison to Other Systems

In this section, we vaguely compare the performance of SmartCOM to other systems. Note that the underlying technology is often different, and the used data sets and parameters are not fully known.

	ETH	Hyperledger
Total time	308439	1510
Mean time	3084	15,1
Median	3132,5	15,5
Max time	6360	25
Min time	363	7

Table 5.17.: Tracing an instance with an increased amount of traces every iteration test metrics. Times in milliseconds.

SEBDB

SEBDB implements a track-trace operation that traces who sends transactions and which transactions are sent. Their tracing operation can trace by the sender, time, and table (also in combination). Additionally, it supports transaction read, update, and write operations. The authors mainly tested their trace and joined operations on a cluster with multiple clients. As their write test measures the scalability with numerous clients, it is not directly comparable. Nevertheless, we can compare SmartCOM with their varying tracking data size [ZZJ+19]. As we do not know if their operation was conducted on a single node or in parallel on multiple, we assume that it was a single node. Their scanning algorithm takes around 10^5 ms for 2000 results, their fastest off-chain indexed algorithm around 80 ms. Consider that our multiple trace algorithm 5.4.4 in the case of Ethereum is a scanning algorithm. In Hyperledger Fabric, the history is recorded. If we extrapolate the 50 traces to test this value, we get for Ethereum a time consumption of 2.4×10^5 ms for 2000 results. For Hyperledger Fabric we re-run our test with 2000 traces of the same object took 830 ms, which is around ten times slower than the off-chain indexed results of SEBDB.

EtherntiyDB

As EthernityDB describes only theoretical costs and time consumption on the main Ethereum network, their results are not comparable (i.e., due to average Block time of 15 seconds). [HRIP18]

BigChainDB

BigChainDB is a Blockchain Database that stores data in MongoDB [Gmb18]. Their results show that without experimental features, their test environment executed 298 transactions per second. As we have no parallel execution test (as the Ethereum nodes need to process serially), we can only look at the other tests. 68% of a test run with one million transactions were executed in under 2,855 seconds [McC18]. We assume that time measurement started with the progressing of a transaction to its end. In our, an expensive single transaction test is a create operation (e.g., test 5.1). These tests take around 2 seconds in Ethereum and Hyperledger, but our test results synchronously are

from local tests. Simultaneously, BigChainDB probably has a much higher latency due to the high number of transactions sent.

Blockchain Meets Database

IBMs Blockchain Meets Database [NGS+19] is a Blockchain Database approach and stores the data in a PostgreSQL database. It shows a latency of under 100 ms when processing 1200 simple contract transactions. As shown in the creation test 5.1 SmartCOM has around 2-second latency on specific operations.

Conclusion from the Comparison

SmartCOM seems to be comparable to basic blockchain database systems. As it does not focus on specific performance improvements, performance-driven blockchain databases will outperform it from such (e.g., indexing of SEBDB). Similarly, to the other blockchain databases with blockchain storage, it is slower than storing the data on the database (as in BigChainDB [Gmb18] or Blockchain Meets Database [NGS+19] approach).

5.6. Conclusion

The proof of concept has shown that SmartCOM can handle different types of blockchain systems. By doing so, SmartCOM managed to introduce the ORM concept to blockchain technology. Although the system might not be a performance improvement or implement new features for the blockchain database system, it allows using existing blockchain databases without blockchain programming knowledge. As these blockchain provider implementations are merely a showcase, it is extendible to the developer's needs. Additionally, if a developer adds a new blockchain provider implementation, all current users can also use the new blockchain system. As some of the architectural features and template file content is similar in both showcases, there might be room to move more content to the parent project (i.e., target API). As we did only two showcases and won't destroy the decoupling between the components, we decided to leave this as a future work option as it would reduce some development time. On the other hand, user management will in most cases differ. Our solution to this problem is that, if a project that uses SmartCOM, it needs to use different implementations of a user class. A user then enters the required information in these implementations. Although this might sound like a tedious process, it gets clear that the user would have to change them nevertheless, somewhere if the credentials change (even if it would be a configuration file). A general configuration file is only sufficient in a one-user solution.

Testing

The tests have shown that some blockchain systems support the idea of a blockchain database better than others. For example, while Hyperledger Fabric supports this idea quite well, Ethereum is struggling with more information as well as tracing. The

implementation of such a provider would nevertheless important, to open more projects to use blockchain storage.

6. Conclusion

In this chapter, we will discuss the limitations of SmartCOM, some possible future work as well as the benefits and drawbacks of SmartCOM.

6.1. Limitation

To keep limit our research question, we decided to limit our project to specific properties. In this section, we discuss these limitations that either became apparent during the creating of SmartCOM or were selected from the start.

6.1.1. Lazy Instance Support

Although some form of lazy instances would be possible to create based on the abstract model interface, it would still be highly dependent on the provider implementation. Consider the case of Hyperledger, where data gets stored in a key-value database. Reading single parts of the value may be only slightly faster than reading all attributes. Nevertheless, connected instance loading on demand would be much faster. The Ethereum provider, on the other hand, would greatly benefit from the lazy instance support both for attributes and especially for connections. As these Attributes change for each POJO, it is not easy to create a generic interface for all providers. A provider-dependent interface would be problematic in terms of maintenance and replaceability of the blockchain provider.

6.1.2. No Support for SQL Like Querying Languages

SmartCOM does not include a query language in any form. All querying will happens over predefined methods (i.e., the auto-generated data repositories), which can be extended when needed. The reason for this is that neither the underlying blockchain systems nor the provided API support a SQL like language. The main benefit of creating an SQL like language nevertheless would be that many developers know SQL.

6.1.3. No Support for Sending Multiple Transactions at Once

As the blockchain provider API typically has no methods to invoke multiple transactions at once, we leave batch sending to the user. The user then can create threads executing single transactions (and can receive results asynchronously). Nevertheless, it has to be kept in mind that blockchain providers like Geth are limited to how many transactions they can accept simultaneously, and the ordering needs to be kept in mind. Due to

serialization problems, even newer blockchain systems such as Hyperchain Ledger tend to abort a transaction if given very large quantities at once [FK21].

6.1.4. No Support for Off-Chain Data

Similarly to [ZZJ+19] we could not guarantee the properties of off-chain data. Additionally, we did not implement any join operations, allowing efficient use of the off-chain data in combination with on-chain data.

6.1.5. Data Migration

Data migration is a problem for blockchain systems as it would require the writing of every transaction in the same order with artificially set senders and timestamps. Such an interface could be invalidated after use (to prevent malicious usage). Our system does not directly support data migration. The reason for this is that in data migration between two different blockchain systems, we would need to have two blockchain providers active via SPI which we currently do not support. But in theory, it would consist of getting all traces, checking the timestamps, and adding the transactions in the same order to the new blockchain system. This would entail that the sender is lost, which could be handled by creating artificial users with the same name in the new system. Additionally, the timestamp would not be accurate (nevertheless, it would follow the same order of events). Finally, data migration between different versions of a contract is not supported. The support of such an operation is dependent on the blockchain provider. In the case of Hyperledger Fabric, it is automatically done on the next write operation (as it overrides old states). Ethereum, on the other hand, would need to have all old addresses of the contracts transferred to the new contract. The problem is that some unknown contracts are referring to the old contract (as not all connections are bi-directional). Finding them would entail checking all possible contracts connected to the old contract and connecting it with the new contract.

6.2. Summary

The main research question of this thesis was whether it is possible to build a blockchain database ORM framework and if it is feasible. The challenges included finding an appropriate architecture to hide a blockchain's complexity, translating user POJOs to smart contracts, and creating smart contract architectures resembling a relational database. We have shown that a ORM framework can be built and extended. The feasibility is highly dependent on the underlying blockchain, the use case, and the blockchain infrastructure. For example, it might be too slow for a website that should immediately show written operation results. Nevertheless, the same problem is also present in other projects using a blockchain as storage. Additionally, we recommend using such a system only in a private blockchain environment as it gets significantly slower and more expensive in a public setting. Besides this, the Hyperledger showcase demonstrated a feasible performance in a private setting, considering the blockchain nature. The Ethereum showcase showed

that techniques like blockchain anchoring might be a better option. Nevertheless, the performance is heavily depending on the provider's implementation. The main focus here is either writing or reading performance (as more complex read operations make contracts larger but can reduce the number of calls needed). Another option is to focus on historic states by writing more information directly into contracts. These options could get implemented as blockchain provider dependencies (e.g., Hyperledger-fast-trace or Hyperledger-fast-write-read).

6.3. Outlook

Possible extensions of SmartCOM would be additional blockchain providers such as Corda [Cor21] or database-based blockchain databases as shown by [JP18]. Although SmartCOM hides the complexity of a blockchain to some extend, there is still the need to configure and start a blockchain manually. An in-memory blockchain test option for testing, research, or learning as a ready-to-use from a dependency would be beneficial (similar to spring-boot with the H2 database). Additionally, more configuration options (similar to the JPA) can be introduced. Support for join operations and off-chain data such as in [ZZJ+19] would also be beneficial. Moreover, SmartCOM could learn from [ZZJ+19] trace parameters as our traces also contain a timestamp and a sender. Another option would be to include blockchain systems in traditional ORM systems such as Hibernate [Hib21]. Additionally, parallelization in ordering or execution steps such as in [NGS+19] would be interesting. As long as blockchains have performance drawbacks, they will be limited to a very restricted set of use cases. Nevertheless, recent development has shown that there is room for improvement. A valid option for some use-cases might be to sacrifice certain blockchain properties to achieve better performance. An example of such a project is LedgerDB which does not implement all blockchain features (i.e., immutability due to the purge method). According to an experimental evaluation, it provides 80 times more throughput than Hyperledger Fabric. LedgerDB [YZW+20].

Bibliography

[AB20] Baeldung Anshul Bansal. Dao vs repository patterns, 2020. Available at `https://www.baeldung.com/java-dao-vs-repository`, Accessed: 17-03-2021.

[ABB+18] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolić, Sharon Weed Cocco, and Jason Yellick. Hyperledger fabric: A distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery.

[AK21a] Petros Demetrakopoulos Alois Klink. Ethair balloons github, 2021. Available at `https://github.com/petrosDemetrakopoulos/ethairballoons`, Accessed: 24-03-2021.

[AK21b] Petros Demetrakopoulos Alois Klink. Ethair balloons smart contract template, 2021. Available at `https://github.com/petrosDemetrakopoulos/ethairballoons/blob/master/lib/contractTemplate.txt`, Accessed: 24-03-2021.

[AMC01] Deepak Alur, Dan Malks, and John Crupi. *Core J2EE Patterns: Best Practices and Design Strategies*. Prentice Hall PTR, USA, 2001.

[Ber05] Max Berger. Data access object pattern. 2005. Available at `https://max.berger.name/research/silenus/print/dao.pdf`, Accessed: 17-03-21.

[bit21a] bitinfocharts. Bitcoin block time historical chart, 2021. Available at `https://bitinfocharts.com/comparison/bitcoin-confirmationtime.html#3m`, Accessed: 19-03-2021.

[bit21b] bitinfocharts. Ethereum block time historical chart, 2021. Available at `https://bitinfocharts.com/comparison/ethereum-confirmationtime.html#3m`, Accessed: 19-03-2021.

[Blo15] Vitalik Buterin (Ethereum Foundation Blog). On public and private blockchains, 2015. Available at `https://blog.ethereum.org/2015/08/07/on-public-and-private-blockchains/`, Accessed: 22-03-2021.

[CCK+18] Mohammad Chowdhury, Alan Colman, Ashad Kabir, Jun Han, and Paul Sarda. Blockchain versus database: A critical analysis. pages 1348–1353, 08 2018.

[CDE+16] Kyle Croman, Christian Decker, Ittay Eyal, Adem Efe Gencer, Ari Juels, Ahmed Kosba, Andrew Miller, Prateek Saxena, Elaine Shi, Emin Gün Sirer, Dawn Song, and Roger Wattenhofer. On scaling decentralized blockchains. In Jeremy Clark, Sarah Meiklejohn, Peter Y.A. Ryan, Dan Wallach, Michael Brenner, and Kurt Rohloff, editors, *Financial Cryptography and Data Security*, pages 106–125, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[com20] European commission. Secure information sharing in federated heterogeneous private clouds (sunfish) project, 2020. Accessed: 14-05-2020.

[Cor21] Corda. Corda homepage, 2021. Available at `https://www.corda.net/`, Accessed: 10-04-2021.

[CRS+18] O. Choudhury, N. Rudolph, I. Sylla, N. Fairoza, and A. Das. Auto-generation of smart contracts from domain-specific ontologies and semantic rules. In *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, pages 963–970, 2018.

[dbl21] dblp. Dblp results, 2021. Available at `https://dblp.org/`, Accessed: 23-03-2021.

[Dem20] Petros Demetrakopoulos. Using data models over ethereum blockchain with ethair balloons, 2020. Available at `https://dev.to/demetrakopetros/using-data-models-over-ethereum-blockchain-using-ethair-balloons-50e9`, Accessed: 24-03-2021.

[Doc21] Hyperledger Docs. Using the fabric test network, 2021. Available at `https://hyperledger-fabric.readthedocs.io/en/release-2.2/test_network.html`, Accessed: 04-04-2021.

[Eth19] Ethereum. Sharding faq, 2019. Available at `https://github.com/ethereum/wiki/wiki/Sharding-FAQ`, Accessed: 14-05-2020.

[Eth21] Etherscan. Blocktime ethereum, 2021. Available at `https://etherscan.io/chart/blocktime`, Accessed: 11-04-2021.

[Eva04] Eric Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley, 2004.

[Fab20] Hyperledger Fabric. Ordering service documentation of hyperledger fabric, 2020. Available at `https://hyperledger-fabric.readthedocs.io/en/latest/orderer/ordering_service.html`, Accessed: 13-05-2020.

[Fab21] Hyperledger Fabric. Hyperledger fabric chaincode java, 2021. Available at `https://github.com/hyperledger/fabric-chaincode-java`, Accessed: 23-03-2021.

[FK21] Dénes László Fekete and Attila Kiss. A survey of ledger technology-based databases. *Future Internet*, 13(8), 2021.

[FN16] Christopher Frantz and Mariusz Nowostawski. From institutions to code: Towards automated generation of smart contracts. pages 210–215, 09 2016.

[Fow02] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Longman Publishing Co., Inc., USA, 2002.

[Gmb18] BigchainDB GmbH. Bigchaindb 2.0 the blockchain database, 2018. Available at `https://www.bigchaindb.com/whitepaper/bigchaindb-whitepaper.pdf`, Accessed: 08-05-2020.

[Gmb20a] BigchainDB GmbH. Bigchaindb 2.0 the blockchain database, 2020. Available at `http://docs.bigchaindb.com/projects/server/en/latest/http-client-server-api.html`, Accessed: 08-05-2020.

[Gmb20b] BigchainDB GmbH. Bigchaindb github, 2020. Available at `https://github.com/bigchaindb/bigchaindb`, Accessed: 12-05-2020.

[GWB91] Richard P. Gabriel, Jon L. White, and Daniel G. Bobrow. Clos: Integrating object-oriented and functional programming. *Commun. ACM*, 34(9):29–38, September 1991.

[Hib21] Hibernate. Hibernate homepage, 2021. Available at `https://hibernate.org/`, Accessed: 10-04-2021.

[HRIP18] Sven Helmer, Matteo Roggia, Nabil El Ioini, and Claus Pahl. Ethernitydb – integrating database functionality into a blockchain. In András Benczúr, Bernhard Thalheim, Tomáš Horváth, Silvia Chiusano, Tania Cerquitelli, Csaba Sidló, and Peter Z. Revesz, editors, *New Trends in Databases and Information Systems*, pages 37–44, Cham, 2018. Springer International Publishing.

[HZD+20] Kai Hu, Jian Zhu, Yi Ding, Xiaomin Bai, and Jiehua Huang. Smart contract engineering. *Electronics*, 9(12), 2020.

[IBNW09] Christopher Ireland, David Bowers, Mike Newton, and Kevin Waugh. Understanding object-relational mapping: A framework based approach. *International Journal on Advances in Software*, 2(2/3):202–216, 2009.

[JK19] Ethan Buchman Jae Kwon. Cosmos: A network of distributed ledgers., 2019. Available at `https://cosmos.network/resources/whitepaper`, Accessed: 08-05-2020.

[JP18] Matthias Kretschmann Jernej Pregelj, Dan Matthews. Crab-based orm for bigchaindb, 2018. Available at `https://github.com/bigchaindb/js-driver-orm`, Accessed: 23-03-2021.

[KP19] Oleksii Konashevych and Marta Poblet. Blockchain anchoring of public registries: Options and challenges. ICEGOV2019, page 317–323, New York, NY, USA, 2019. Association for Computing Machinery.

[LGGBB18] Faiza Loukil, Chirine Ghedira-Guegan, Khouloud Boukadi, and Aïcha Nabila Benharkat. Semantic iot gateway: Towards automated generation of privacy-preserving smart contracts in the internet of things. In Hervé Panetto, Christophe Debruyne, Henderik A. Proper, Claudio Agostino Ardagna, Dumitru Roman, and Robert Meersman, editors, *On the Move to Meaningful Internet Systems. OTM 2018 Conferences*, pages 207–225, Cham, 2018. Springer International Publishing.

[LHR16] M. Lorenz, Guenter Hesse, and J. Rudolph. Object-relational mapping revised - a guideline review and consolidation. In *ICSOFT-EA*, 2016.

[McC18] Troy McConaghy. And we're off to the races!, 2018. Available at `https://blog.bigchaindb.com/and-were-off-to-the-races-1aff2b66567c`, Accessed: 10-04-2021.

[MRK+21] Emiliano Monteiro, Rodrigo Righi, Rafael Kunst, Cristiano da Costa, and Dhananjay Singh. Combining natural language processing and blockchain for smart contract generation in the accounting and legal field. In Madhusudan Singh, Dae-Ki Kang, Jong-Ha Lee, Uma Shanker Tiwary, Dhananjay Singh, and Wan-Young Chung, editors, *Intelligent Human Computer Interaction*, pages 307–321, Cham, 2021. Springer International Publishing.

[NGS+19] Senthil Nathan, Chander Govindarajan, Adarsh Saraf, Manish Sethi, and Praveen Jayachandran. Blockchain meets database: Design and implementation of a blockchain relational database, 2019.

[Ors06] Jaroslav Orság. *OBJECT-RELATIONAL MAPPING*. dissertation, Comenius Univerity, 2006.

[Pre17] Jernej Pregelj. Crab — create. retrieve. append. burn., 2017. Available at `https://blog.bigchaindb.com/crab-create-retrieve-append-burn-b9f6d111f460`, Accessed: 23-03-2021.

[Smi17] Anton Smirnov. Web3j issue: web3j is too slow, 2017. Available at `https://github.com/web3j/web3j/issues/231`, Accessed: 11-04-2021.

[Sol20] Soliditylang. Types, 2020. Available at `https://docs.soliditylang.org/en/latest/types.html`, Accessed: 28-03-2021.

[SP19] Qiuyun Shang and Allison Price. A blockchain-based land titling project in the republic of georgia: Rebuilding public trust and lessons for future pilot projects. *Innovations: Technology, Governance, Globalization*, 12:72–78, 01 2019.

[Sta18] Starchain. Starchain global digital asset chain in culture and entertainment, 2018. Available at `https://kp-oss-test.oss-cn-shenzhen.aliyuncs.com/stc/StarChain-White-paper.pdf`, Accessed: 11-06-2020.

[Ten20] Tendermint. Tendermint broadcast api, 2020. Available at `https://docs.tendermint.com/master/tendermint-core/using-tendermint.html#broadcast-api`, Accessed: 08-05-2020.

[TYSS19] T. Tateishi, S. Yoshihama, N. Sato, and S. Saito. Automatic smart contract generation using controlled natural language and template. *IBM Journal of Research and Development*, 63(2/3):6:1–6:12, 2019.

[XF21] Peichang SHI Xiang FU, Huaimin WANG. A survey of blockchain consensus algorithms: mechanism, design and applications. *SCIENCE CHINA Information Sciences*, 64(2), 2021.

[XZLH20] Y. Xiao, N. Zhang, W. Lou, and Y. T. Hou. A survey of distributed consensus protocols for blockchain networks. *IEEE Communications Surveys and Tutorials*, 22:1432–1465, 2020.

[YZW+20] Xinying Yang, Yuan Zhang, Sheng Wang, Benquan Yu, Feifei Li, Yize Li, and Wenyuan Yan. Ledgerdb: A centralized ledger database for universal audit and verification. *Proc. VLDB Endow.*, 13(12):3138–3151, August 2020.

[Zhe18] Blockchain challenges and opportunities: A survey. *Int. J. Web Grid Serv.*, 14(4):352–375, January 2018.

[ZZJ+19] Y. Zhu, Z. Zhang, C. Jin, A. Zhou, and Y. Yan. Sebdb: Semantics empowered blockchain database. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 1820–1831, 2019.

Acronyms

API Application Programming Interface. 10, 43, 63, 68, 81

AST Abstract Syntax Tree. 31, 32

BFT Byzantine Fault Tolerance. 5, 14, 21, 23

CRUD Create Read Update Delete. iii, 28–30, 39, 46, 53, 62

CSV Comma Seperated Values. 57

DAO Data Access Object. xiii, 10, 11

DSL Domain Specific Language. 30, 32

ECTS European Credit Transfer System. 67

ETH Ethereum. 71–78

HTTP Hypertext Transfer Protocol. 68

IoT Internet of Things. 31

JPA Java Persistence API. 29, 34, 36, 47, 83

JSON JavaScript Object Notation. 37, 48, 68–70

MDA Model Driven Architecture. 30, 32

NLP Natural Language Processing. 31, 32

NoSQL Not Only Structured Query Language. 16

OOPL Object Oriented Programming Language. 9, 10, 48

ORM Object Relational Mapping. iii, v, 1, 2, 5, 9, 12, 13, 29, 30, 48, 79, 82, 83

PBFT Practical Byzantine Fault Tolerance. 7

POJO Plain Old Java Object. xiii, 29, 33, 34, 36–40, 44, 47–49, 51–53, 55, 57, 59, 60, 62, 63, 68–70, 81, 82

PoS Proof of Stake. 7

PoW Proof of Work. 6, 7, 16

REST Representational State Transfer. ix, xiii, 45, 48, 68, 69

RPC Remote Procedure Call. 43, 49, 68

SEBDB Semantics Empowered BlockChain DataBase. 17

SI Snapshot Isolation. 14, 15

SmartCOM Smart Contract Object Mapping. iii, v, viii, xiii, 1–3, 5, 13, 33–36, 38, 40–49, 52–54, 57, 63, 67–69, 77–79, 81, 83

SPI Service Provider Interface. 37–39, 43, 44, 46, 82

SQL Structured Query Language. iii, v, ix, 16, 22, 28, 81

SSI Serializable Snapshot Isolation. 14, 15, 24

TPS Transactions Per Second. 20

URL Uniform Resource Locator. 37, 44

UTXO Unspent Transaction Output. 20

VTL Velocity Template Language. xv, 37, 44–46, 63

XML Extensible Markup Language. 12

YAML YAML Ain't Markup Language. xv, 42, 54, 55, 66

Glossary

chaincode the name for Hyperledger fabric smart contracts. Can be written in multiple programming languages as for example Java. 29, 59, 60, 68

cryptocurrency wallet Program or physical medium that stores public and private keys needed to control cryptocurrencies. 6, 22, 48, 54, 57

Ethereum An open-source decentralized blockchain with smart contract capabilities. viii, xiii, 5, 7–9, 16, 18–20, 25, 26, 30, 34, 48–50, 53, 67, 68, 70–79, 81

Genesis file In Ethereum, the genesis file is the start of the blockchain. It describes multiple parameters, for example, the mining difficulty. 48

Hyperledger Fabric a distributed ledger framework. Running different types of smart contracts in a containerized manner. ix, xiii, 18–20, 28, 29, 31, 45, 59–61, 63–67, 78, 79

Java connector A Java class able to call functions of a related Solidity class on the blockchain. For example, contacting a delete method in a Java connector for a table, would trigger a delete function in the Solidity table contract. 53, 63

smart contract address An address is uniquely referring to an instance of a smart contract. Unique means if the first user contract gets generated, it will get for example, the address 0x123. If the blockchain gets asked for the smart contract at 0x123, it will return user number one. 51, 52, 54, 57, 66

Solidity Object-oriented programming language for implementing smart contract running on Ethereum. viii, xiii, 19, 26, 30, 36, 45, 48, 49, 51–58, 67

A. Appendix

A.1. Framework Startup Description

How to Run Solidity

- Install Maven
- Install Java JDK 11+
- Solidity compiler renamed to "solc" (`https://github.com/ethereum/solidity/releases/tag/v0.7.0`) added to Path
- Install web3j-cli version 1.4.0+ `https://github.com/web3j/web3j-cli/releases/tag/v1.4.1`
- Install GETH (preferably version 1.9.24) `https://geth.ethereum.org/downloads/`
- Add Solidity, Web3j and GETH to the Path:
  - Unix (if permanent in .bashrc): `export PATH="$HOME/bin:$PATH:DOWNLOAD_LOCATION"`
  - Windows: add the Solidity compiler location (renamed to solc.exe), web3j (web3j.exe) binary and GETH (geth.exe) in the Environment Variables
- Change directory to `01516200-martin-pfitscher/prototype/geth/blockchain/test`
- Keep the keystore in the data location, instantiate the data location with "geth –datadir=data init genesis.json"
- Start GETH with `geth -http -http.addr 0.0.0.0 -networkid 1234 -port 30303 -http.port 8545 -datadir data -http.api="eth,net,web3,personal" -http.corsdomain "*" -syncmode fast -cache=4096`
- Now connect with the ipc in a new terminal window (the address is should be printed in the console otherwise try `geth attach \.\pipe\geth.ipc`): `geth attach PRINTED_ADDRESS`
- In the console run `miner.start()` you can check on the allocated ether with `eth.getBalance(eth.coinbase)`

- Navigate to the prototype folder (`cd /01516200-martin-pfitscher/prototype`)
- Command in terminal: mvn clean install package
- All sources should be installed by now (and test should run through if not disabled)

How to Run Hyperledger

1. Install Maven
2. Install Java JDK 11+
3. Make sure Docker is running
4. Change directory to 01516200-martin-pfitscher/protoype
5. Install Hyperledger fabric at this location "chmod +x bootstrap.sh" and then "./bootstrap.sh -s"
6. Command in terminal: `mvn clean install package`
7. Start the network with: ./costumStart.sh (requires super user rights e. g. sudo ./costumStart.sh)
8. Run "mvn clean install" again to run the tests

How to Run the REST Demo

1. Have all the dependencies for the Hyperledger or Solidity implementation depending on what you are using.
2. Start your Blockchain system.
3. Run the `maven command spring-boot:run` in the rest project.
4. Wait until finished starting, then import the *.postman_collection.json commands into postman
5. Run the commands against the system or modify the JSONs sent

A.2. Performance Test Figures

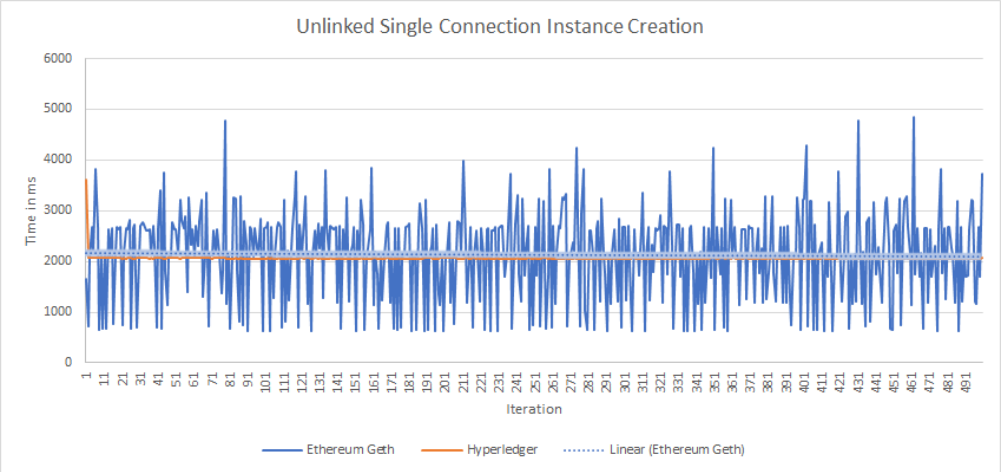

Figure A.1.: Time consumption per iteration in single empty connections creation test

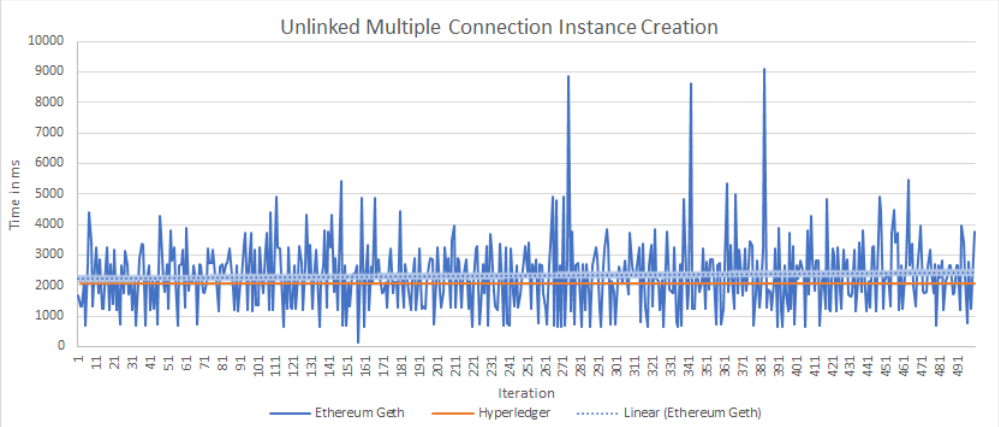

Figure A.2.: Time consumption per iteration in multiple empty connections creation test

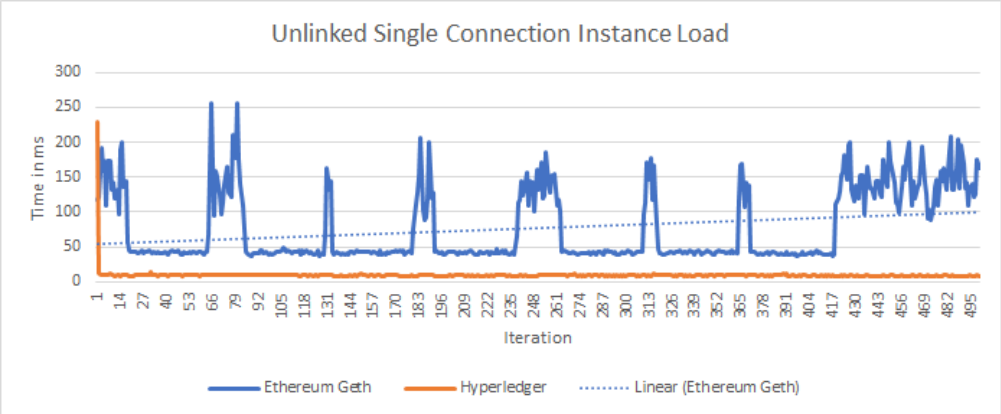

Figure A.3.: Time consumption per iteration in single empty connections loading test

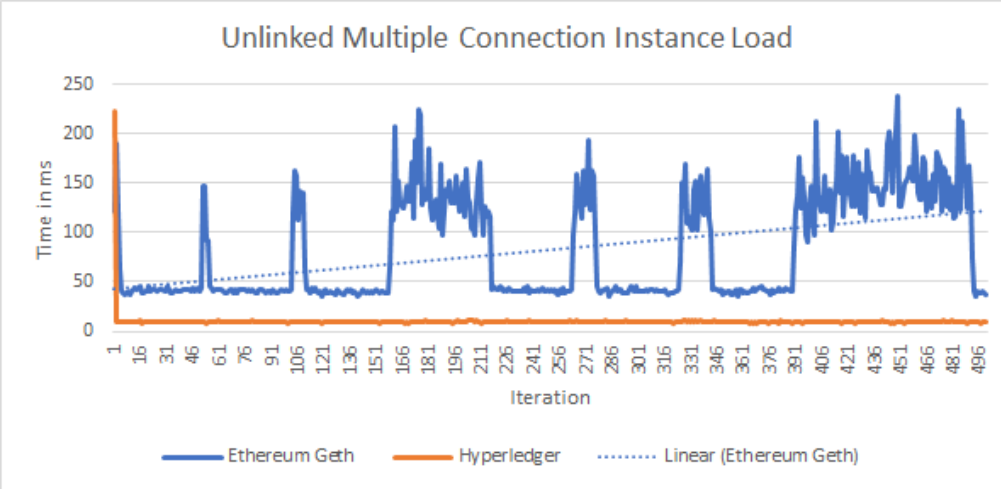

Figure A.4.: Time consumption per iteration in multiple empty connections loading test

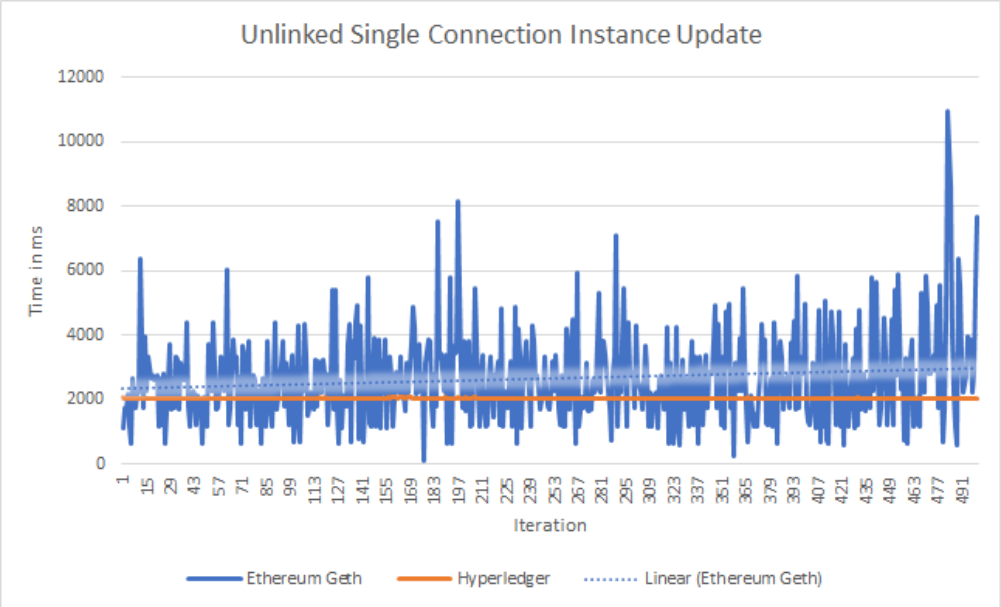

Figure A.5.: Time consumption per iteration in single instance update test

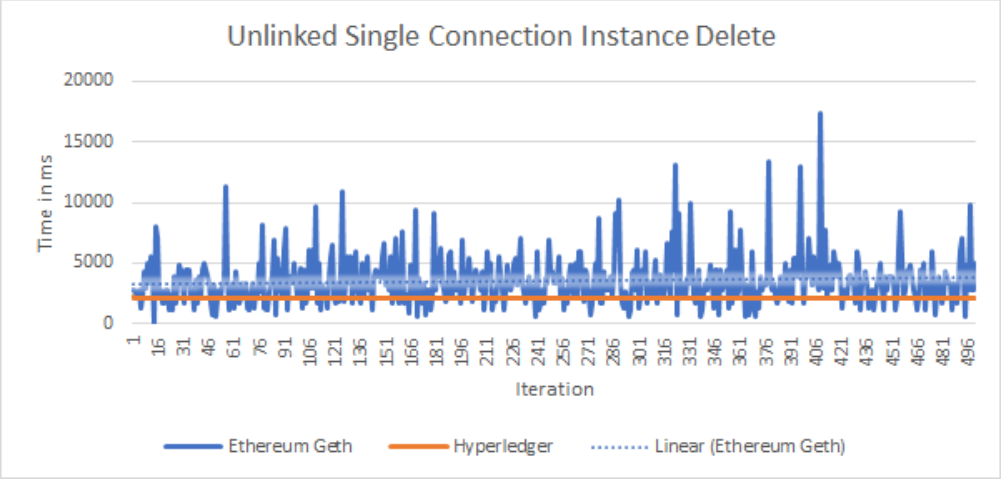

Figure A.6.: Time consumption per iteration in single instance delete test

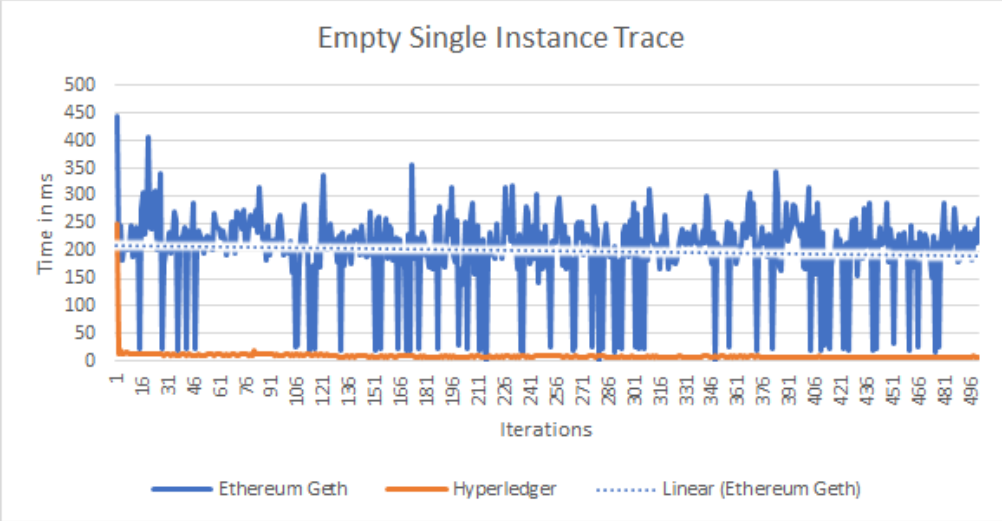

Figure A.7.: Time consumption per iteration in single instance trace test

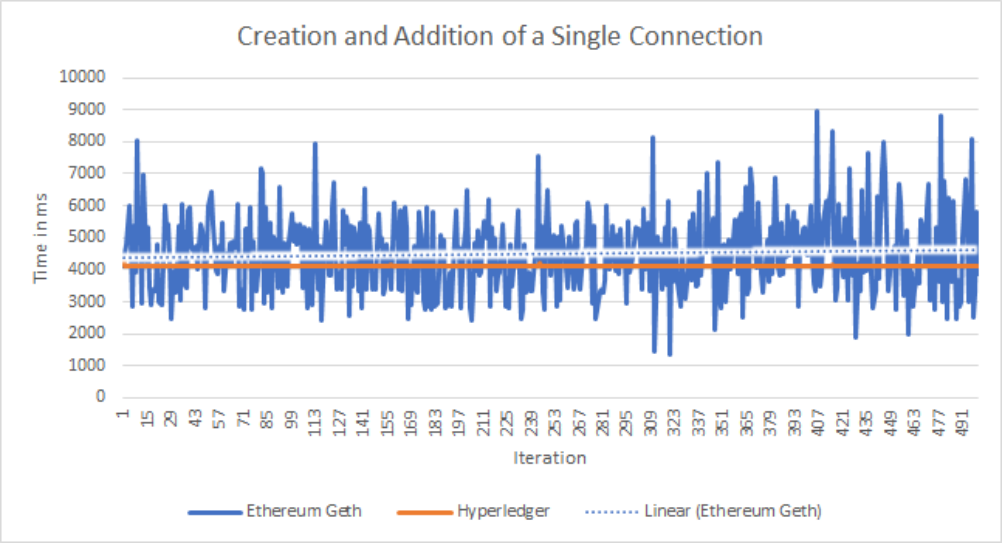

Figure A.8.: Time consumption per iteration single connection creation test

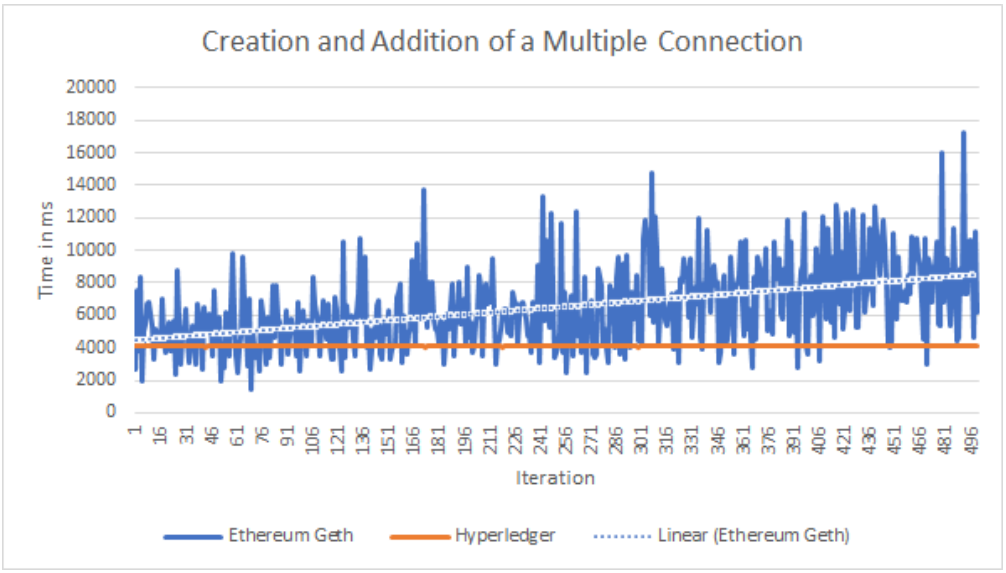

Figure A.9.: Time consumption per iteration multiple connection creation test

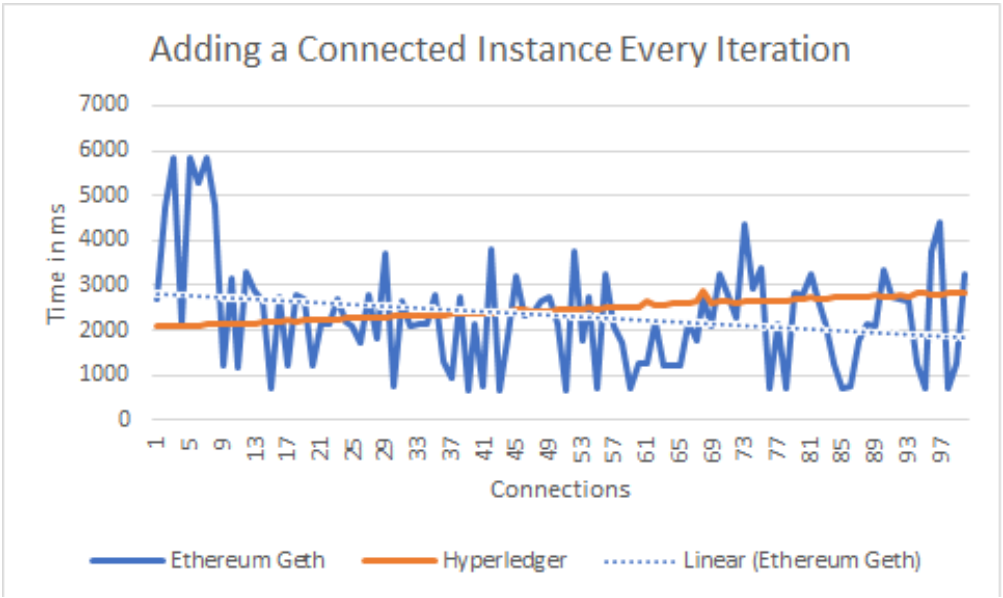

Figure A.10.: Time consumption creating one instance with number of iterations connections every iteration test

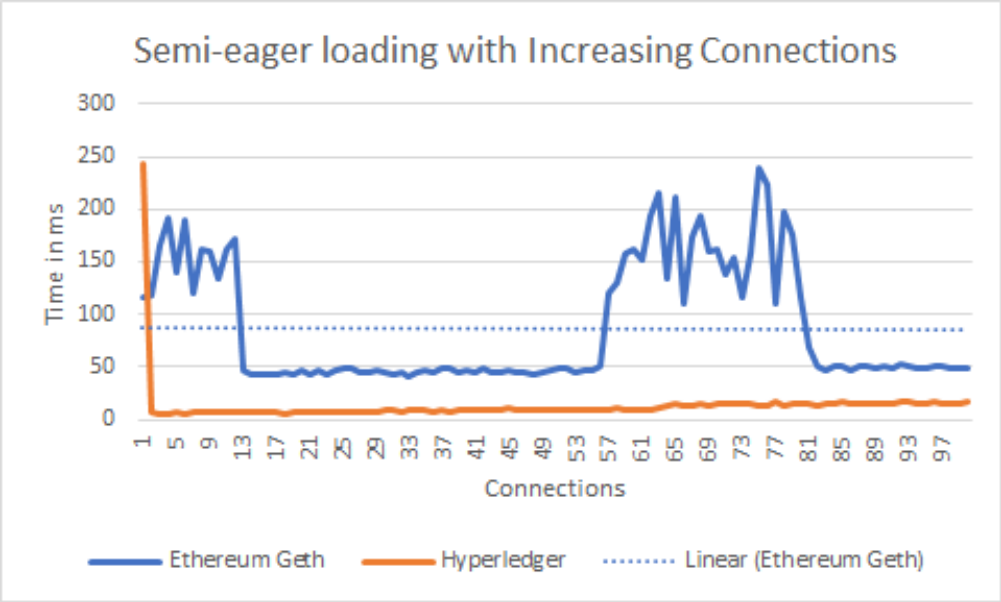

Figure A.11.: Time consumption per iteration semi-eager load test of instances with increasing connections

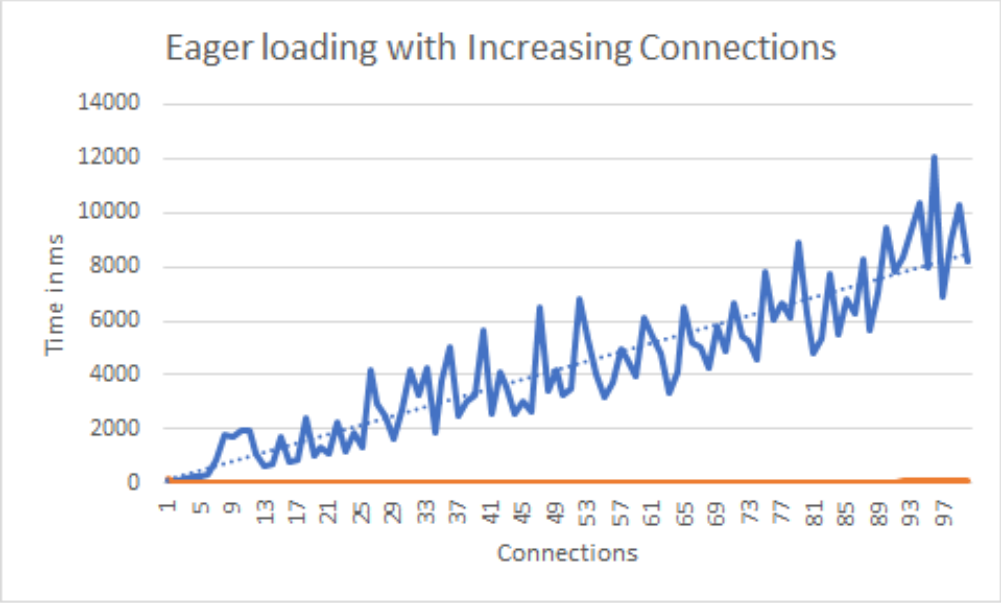

Figure A.12.: Time consumption per iteration eager load test of instances with increasing connections

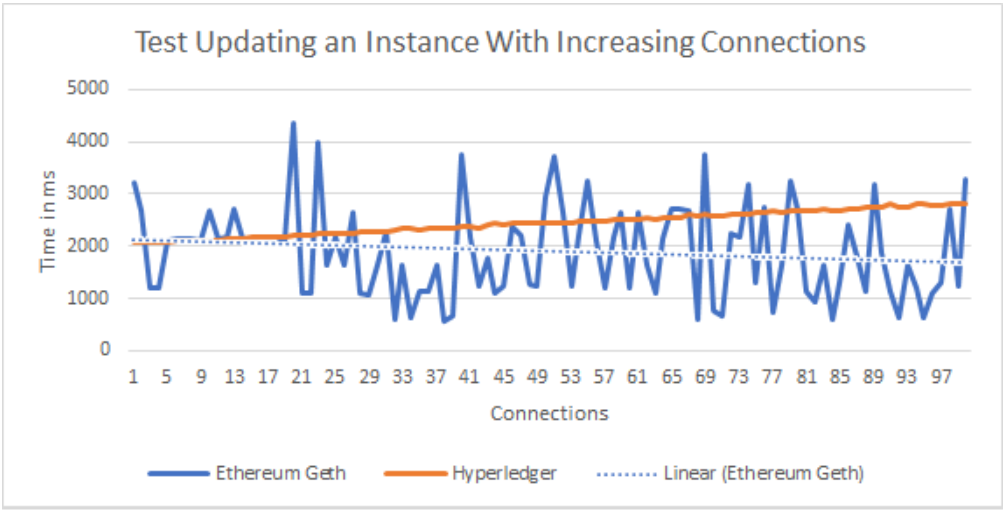

Figure A.13.: Time consumption updating a instance with number of iteration connections test

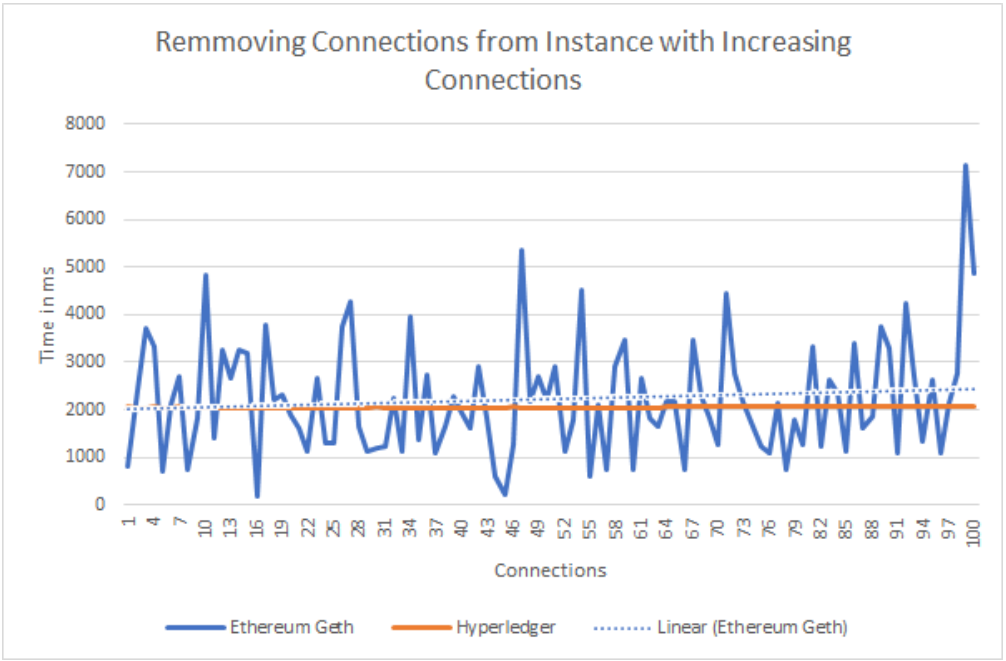

Figure A.14.: Time consumption removing all connections from a instance with number of iteration connections test

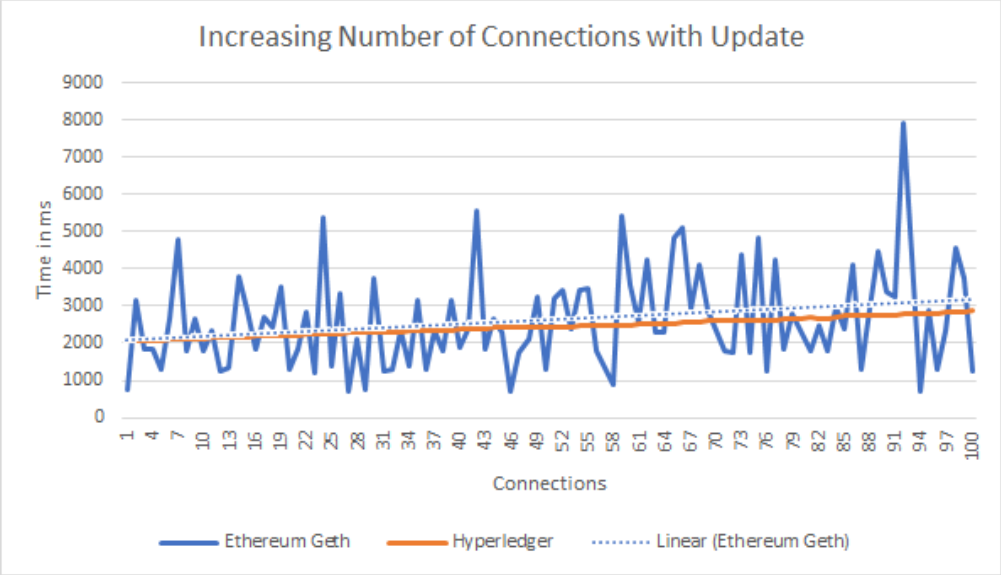

Figure A.15.: Time consumption adding one instance via update every iteration to the multiple connection instance test

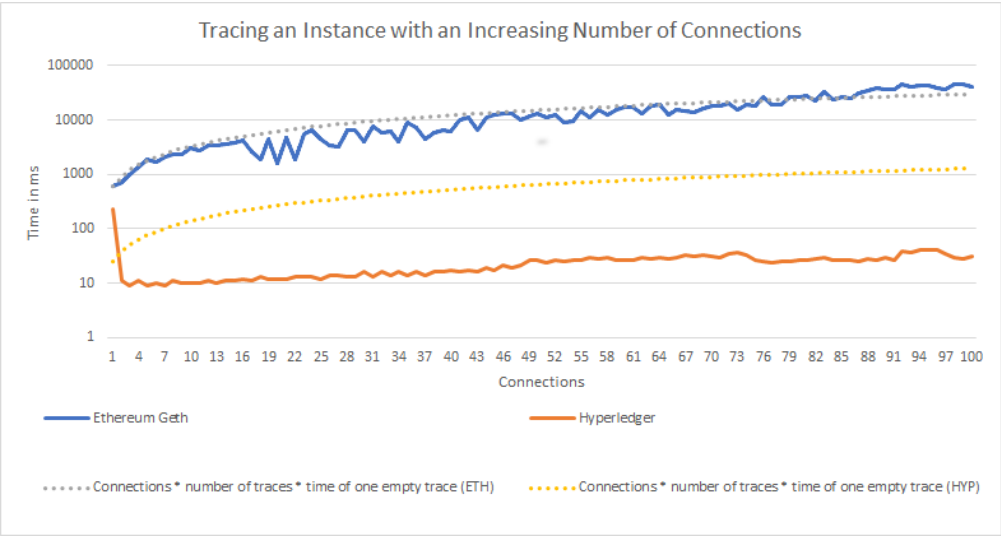

Figure A.16.: Time consumption tracing a instance with a increased amount of connections every iteration to the test

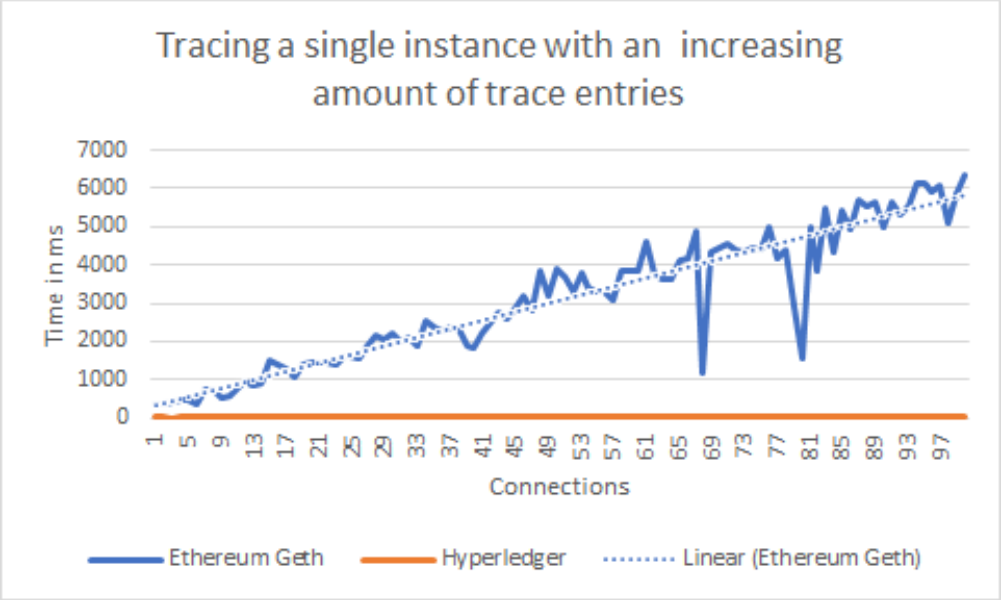

Figure A.17.: Time consumption tracing a instance with a increased amount of traces every iteration to the test