FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

# Community Blockchain Interaction Patterns

DIPLOMARBEIT

zur Erlangung des akademischen Grades

**Diplom-Ingenieurin**

im Rahmen des Studiums

**Software Engineering & Internet Computing**

eingereicht von

**Martina Sengstschmid, BSc**
Matrikelnummer 00707282

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: A.o. Univ. Prof. Dr. Dipl.-Ing. Eva Maria Kühn
Mitwirkung: Dr. Dipl.-Ing. Gerson Joskowicz

Wien, 4. Juni 2019

| Martina Sengstschmid | Eva Maria Kühn |
|---|---|

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

# Community Blockchain Interaction Patterns

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

**Diplom-Ingenieurin**

in

**Software Engineering & Internet Computing**

by

**Martina Sengstschmid, BSc**
Registration Number 00707282

to the Faculty of Informatics

at the TU Wien

Advisor: A.o. Univ. Prof. Dr. Dipl.-Ing. Eva Maria Kühn
Assistance: Dr. Dipl.-Ing. Gerson Joskowicz

Vienna, 4th June, 2019

Martina Sengstschmid

Eva Maria Kühn

# Erklärung zur Verfassung der Arbeit

Martina Sengstschmid, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 4. Juni 2019

Martina Sengstschmid

# Danksagung

An dieser Stelle möchte ich meiner Betreuerin Eva Maria Kühn und meinem Betreuer Gerson Joskowicz für ihre fortwährende Unterstützung ganz herzlich danken. Ihr konstruktives Feedback in zahlreichen Gesprächen hat mir den gesamten Entstehungsprozess dieser Arbeit hindurch geholfen. In diesem Zusammenhang auch danke an Marion, welche viele dieser Gespräche positiv bereichert hat.

Abschließend möchte ich mich bei meinem Freund, meiner Familie, und meinen Freunden für ihre Unterstützung, ihr Verständnis und ihren stets positiven Zuspruch in dieser Zeit bedanken.

# Acknowledgements

At this point, I would like to thank my supervisors Eva Maria Kühn and Gerson Joskowicz for their continuous support. Their constructive feedback in numerous discussions helped me through the whole process of creating this work. In this context, also thanks to Marion for her positive contribution to these conversations.

Finally, I would like to thank my boyfriend, family, and friends for their support, understanding, and positive encouragement during this time.

# Kurzfassung

Ausgelöst durch den Kryptowährungs-Boom der letzten Jahre erfreut sich die darunterliegende Blockchain-Technologie größter Beliebtheit. Dies liegt nicht zuletzt an den inhärenten Eigenschaften der Technologie, wie beispielsweise Dezentralisierung, Transparenz, Unveränderlichkeit der Daten, öffentliche Überprüfbarkeit, Nicht-Abstreitbarkeit, Integrität, Datenschutz und gleiches Recht für alle. Die Blockchain ermöglicht verteilte Datenhaltung zwischen nicht vertrauenswürdigen Parteien in einem verteilten Netzwerk, ohne die Notwendigkeit einer zentralen Instanz.

Angetrieben von den vielversprechenden Möglichkeiten investieren viele Firmen unsummen in neue Blockchain Projekte. Dadurch stieg die Anzahl an neuen Blockchain Systemen, allen voran Permissioned Blockchains. Diese haben die Eigenschaft, den Netzwerkzugriff zu beschränken, wodurch sie vor allem für den Einsatz im Firmenbereich gut geeignet sind. Mit der steigenden Anzahl verschiedener Systeme wird es zunehmend schwerer für alle Anforderungen eine geeignete Blockchain-Technologie zu finden.

Diese Arbeit illustriert eine pattern-basierte Methodik zur Evaluierung eines bestimmten Blockchain-Ansatzes für eine Anwendung. Ausgehend von kollaborativen Community Anwendungen und der dazugehörigen Use Cases werden sechs Community Blockchain Interaction Patterns abgeleitet. Die Arbeit zeigt weiters, wie sich bestehende Permissioned Blockchain Systeme anhand der Patterns analysieren lassen. Das Ergebnis dieser Analyse zeigt deutliche Unterschiede bei der Unterstützung der Patterns. Abschließend wird gezeigt, dass die entwickelten Patterns in Anwendungen aus verschiedensten Industriezweigen anwendbar sind.

# Abstract

Encouraged by the cryptocurrency boom of recent years, the underlying blockchain technology enjoys great popularity. This is reasoned by the inherent properties of the technology, such as decentralization, transparency, immutability, public verifyability, non-repudiation, integrity, privacy and equal rights. The blockchain enables distributed data management between untrusted parties in a distributed network, without the need of a central authority.

Driven by the promising opportunities, many companies are investing large sums into new blockchain projects. This increased the number of new blockchain systems, especially permissioned blockchains. Permissioned blockchains restrict the access to the network, making them particularly well suited for use in the corporate sector. As the number of systems increases, it becomes more difficult to find a suitable blockchain technology for all requirements.

This thesis illustrates a pattern-based methodology for evaluating the suitability of a particular blockchain approach for a given application. Based on collaborative community applications and the associated use cases, six community blockchain interaction patterns are derived. Furthermore, the work shows how existing permissioned blockchain systems can be analyzed using these patterns. The result of this analysis shows significant differences in how well the systems support the implementation of the patterns. Finally, it is shown that the proposed patterns can be used in applications across various industries.

# Contents

CHAPTER 1

# Introduction

## 1.1 Motivation

The hype around Bitcoin, Ethereum and other cryptocurrencies caused great interest in the underlying blockchain technology, which is also known as distributed ledger technology (DLT). Both, academia and companies are driven by the big promises of the blockchain. This is mainly based on the inherent principles of the technology. The blockchain establishes high trust and transparency between untrusted parties in a fully decentralized network, without the need of a central authority or third party service. In 2008, Satoshi Nakamoto published a whitepaper [Nak08], introducing the first decentralized cryptocurrency called Bitcoin. It enables transfer of digital currency in an untrusted distributed peer-to-peer (P2P) network. Bitcoin is the first real cryptocurrency that solves the double spending problem by the use of a new Proof of Work (PoW) consensus algorithm. This PoW describes an agreement process on the correct state of a data ledger, which is shared between untrusted parties in the distributed network. Inspired by Bitcoin, the Ethereum blockchain was introduced in 2014 by Vitalik Buterin [But14]. It also uses the PoW algorithm, but additionally provides smart contracts to integrate complex business logic. Smart contracts are programs, that are stored in the blockchain ledger and can be executed by the participants in a blockchain network. Soon, multiple companies initiated blockchain projects to investigate the applicability of the blockchain technology on their business processes. However, true business applications require more flexibility, privacy, and complexity than Bitcoin or Ethereum can provide. Therefore, many companies started to develop their own enterprise blockchain system. This led to an enormous number of different blockchain implementations.

A comparison of the systems is difficult, since there is still no standardization for blockchains. To find the best solution for a particular scenario, the pros and cons of each blockchain system need to be evaluated. The differences in consensus mechanisms and the complexity of the systems make this a tedious task. Some approaches, like [WG18],

provide a guide to decide which kind of blockchain (public or permissioned) to use. Apart from this rather crude classification, companies still need to analyse the requirements of each use case separately in order to select a suitable blockchain system.

## 1.2 Problem Statement

Companies have specific requirements regarding privacy and transparency of data. Within a company, there is usually a certain relationship of trust. In different departments, the employees work closely together in small groups. Members of such groups must independently coordinate and organize their work. In some cases, sensitive information is exchanged between employees within a group. In other cases, the privacy of individual group members is important to ensure good cooperation (i.e., when group members need to find an agreement). It should be possible to establish trust between group members fully autonomous and without the need of a central authority. Similar assumptions also apply to cross-company cooperation. The term "community" is appropriate, whenever a group of people needs to collaborate to pursue a specific goal. According to [WG18] and [ZXD+17], some of the inherent benefits of the blockchain technology are decentralization, transparency, privacy, integrity, and public verifiability. Since, the collaboration in groups requires similar properties, the blockchain technology can be used to create distributed community applications. Such a blockchain is called "community blockchain". It provides the required trust between members of a community, without the need of a central third party. Data privacy and transparency are important in this context. In the context of this thesis, the notion of "community" assumes a basic level of trust between the participants, which extends to the identity of the particpants but not necessarily to the processes implemented on the community-blockchain. In contrast to public blockchains, permissioned blockchains restrict the set of participants that are allowed to interact within the blockchain network. Therefore, permissioned blockchains are well suited as a basis for the implementation of community use cases. But with the growing number of blockchain systems, it becomes increasingly difficult to select the most suitable solution for all applications.

## 1.3 Aim of the Work

The expected outcome of this thesis yields a set of patterns for community blockchains (for small to mid-size groups in business environments). This thesis pursues a classification approach, where these community blockchain patterns are used to identify important requirements in community processes. Furthermore, the patterns can be used to identify suitable permissioned blockchain systems and show both deficiencies and advantages in respect to the described scenarios. Thererfore, the envisioned solution supports a pattern-based analysis of existing permissioned blockchain systems. This methodology allows software developers to investigate the suitability of permissioned blockchain systems in such environments. Specific issues in collaborative processes, like privacy and transparency, need to be considered.

## 1.4 Methodological Approach

First, different already existing permissioned blockchain systems (e.g., Hyperledger Fabric, Corda, Tendermint) are examined and some of them are selected for the pattern-based analysis. The systems are described briefly, giving an introduction to their components, transactions, consensus mechanisms, contracts, and permission handling. The description and selection process is based on information provided by whitepapers and the official documentation of the systems.

Next, typical community blockchain use cases for collaborative processes on the blockchain are identified and described.

Afterwards, collaborative blockchain interaction patterns are derived from the outlined community blockchain use cases.

Next, the previously described existing permissioned blockchain solutions will be analysed in respect to their ability to implement the patterns. A methodology to describe their architectural flexibility according to new requirements is needed for classification.

Then, the results of the analysis are outlined and discussed. The discussion shall highlight the advantages and disadvantages of existing permissioned blockchain systems.

Finally, the universal applicability of the outlined patterns is shown. Therefore, potential blockchain use cases from different application areas are analysed using the described patterns.

## 1.5 Structure of the Work

This thesis consists of eight chapters. It starts with a general introduction in chapter 1, that includes the motivation for this work as well as the problem statement, aim, the used methodology and the structure. Chapter 2 provides an overview on the basic principles of the blockchain technology, namely the peer-to-peer network, the blockchain ledger, transactions, consensus, (smart) contracts and permissions. The chapter is concluded by summarizing blockchain specific properties. The related work, including different taxonomy approaches, summaries on potential blockchain applications and current research on blockchain patterns is described in chapter 3. In chapter 4 a selection of six of the most commonly used permissioned blockchain systems is described on basis of their published whitepapers and documentation. Furthermore, other permissioned blockchain systems that were examined in the selection process, are briefly described. Chapter 5 contains eight community blockchain use cases that require close collaboration between the stakeholders. Based on these use cases, six community blockchain interaction patterns are derived and outlined in this chapter. The relation between the use cases and patterns is described for traceability reasons. In chapter 6 the previously selected permissioned blockchain systems are analysed based on the outlined patterns. The results of this analysis are summarized and discussed in chapter 7. Furthermore, the universal applicability of community blockchain interaction patterns is shown based on

blockchain applications from different industries. Chapter 8 concludes this work with a short summary and outlines potential future work.

CHAPTER 2

# Blockchain Basics

The technology behind Bitcoin [Nak08] and other cryptocurrencies is called blockchain or distributed ledger technology (DLT). A blockchain is an appendable-only data structure, that is managed and replicated in a decentralized peer-to-peer (P2P) network. The data is completely immutable once stored on the blockchain, in contrast to traditional distributed database systems. Furthermore, there is no need for a central authority or administrator to establish trust between the participants (peers) of the network. Instead, the peers agree on the correct state of the data using complex consensus mechanisms. They ensure that the state is correct even under adverse conditions when some of the participants are fraudulent or disconnected (Byzantine Failures).

The mechanics of blockchain technology are best described by the steps performed in the blockchain network [Nak08]: To add new data to the blockchain, a peer creates and signs a transaction, which embodies a proposed change of the overall system state. This transaction is published to the blockchain network, where it is broadcasted to all peers. Each peer verifies the data and signature of the transaction. Inconsistent transactions are discarded immediately. Successful verified transactions get wrapped into a block. The creator of a block calculates the hash value of the previous block and includes this hash into the header of the new block. This hash guarantees immutability of the ledger, meaning that any modification of data in the ledger will be detected immediately by the other peers. Once a block is created, the creator propagates it to the network. The other peers verify the correctness of the data in the block and agree to include that block as next block in their local copy of the blockchain. Every peer sees all data included in the block, therefore the ledger supports full **transparency**. The agreement process is called consensus algorithm and establishes the consented "truth" according to rules adapted to the type of blockchain (e.g., longest chain or majority vote).

The consensus mechanism in a blockchain network conforms to a democratic-decision making process. The utilization of cryptographic algorithms for signing and verification and the replication in a distributed network guarantees data **integrity** in the blockchain. The

blockchain technology combines decentralized peer-to-peer networks with sophisticated cryptographic concepts. This enables a high level of transparency and verifiability.

In this chapter, the basic concepts of the blockchain technology are outlined in more detail. To better understand these principles, they are explained using popular representatives for the blockchain technology, namely Bitcoin and Ethereum. Finally, basic properties of blockchains are outlined.

## 2.1 P2P Network

Most business applications and cloud solutions are based on a centralized network approach. In a centralized system, data storage and communication is coordinated by a single central authority (server). The communication is based on request-response patterns. Clients send requests to the server and wait for a response. This approach has the drawback that one central server holds all information, and a malfunction of the server can lead to information loss. A centralized configuration is not resistant to network disconnections and represents a single point of failure. It is often targeted first by attackers in order to harm the system, for example in a denial of service (DoS) attack. If the central unit stops working, the whole system fails. As centralized systems scale out, their complexity increases dramatically through demands on internal replication, load balancing and network capacity. P2P networks tempt to address these problems in a **decentralized** manner, where clients communicate directly with each other. Figure 2.1 shows the different topologies for centralized (client-server) and distributed (P2P) networks.

A precise definition of a peer-to-peer network was given by Schollmeier in 2001 [Sch01]:

> "A distributed network architecture may be called a Peer-to-Peer (P-to-P, PZP, ...) network, f the participants share a part of their own hardware resources (processing power, storage capacity, network link capacity, printers, ...). These shared resources are necessary to provide the Service and content offered by the network (e.g. file sharing or shared workspaces for collaboration): They are accessible by other peers directly, without passing intermediary entities. The participants of such a network are thus resource (Service and content) providers as well as resource (Service and content) requestors (Servent-concept)."[1]

This means, that a P2P network allows a peer to find another peer through required resource properties other than network addresses. Although peers know initially only the network addresses of some of their immediate neighbours, they are able to find peers within the whole, dynamically changing P2P network. Public blockchain networks rely on this capability, which is implemented by distributed hash table (DHT) technologies.

[1] [Sch01] page 1.

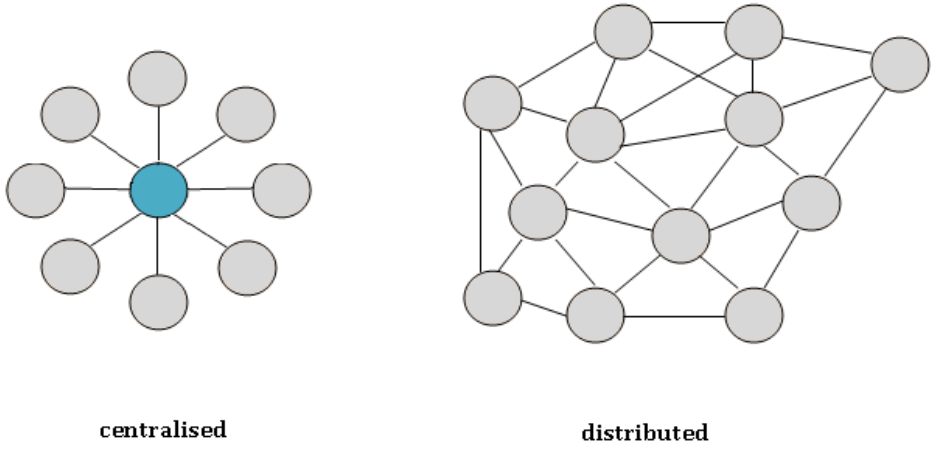


Figure 2.1: Network topology of centralised and distributed systems. A single server maintains all communication (left) between peers in a centralised system, while each peer in a distributed system is connected to several other peers (right).

One of the earliest P2P networks was introduced by Shawn Fanning in 1999 as a decentralized client software for music and file-sharing called Napster. The topology still involved a central server for the file lookup, but the actual file-transfer happened between two clients without the need of an intermediary. Another file-sharing network called Gnutella was introduced in 2000. In contrast to Napster, it was designed to work without any central authority. With these popular applications that supported illegal multiplication of copyrighted music, the P2P technology got a bad reputation in public. Since then, blockchain is the first technology using a P2P network, that is dissociated from this rather negative image.

The basic idea of Bitcoin is to use a distributed P2P network, where everyone can join and has **equal rights**. Each participant can act anonymously and without any precondition, i.e. without any predefined notion of trust between participants. Furthermore, every participant contributes memory and processing power to enable trust for the reliable transfer of monetary values in an untrusted network. There are different implementations of the Bitcoin protocol, which provides a solution to the double spending problem. In the Bitcoin Core implementation, every peer has four main functions. It maintains a full copy of the **replicated** blockchain ledger, holds a wallet (account to send and receive Bitcoins), performs verification and populates valid transactions and blocks in the network, and actively participates in the block creation process as a miner [Ant14]. The miner is the nexus of the Nakamoto consensus: The mining process creates an incentive to participate in the distributed verification process, it creates bitcoins and it wards off

malicious users. In practice this vision turns out to be impractical. A peer might not have the resources to store hundreds of gigabytes of data to maintain a full copy of the blockchain ledger. Bitcoin's blockchain for example has at the time of writing already around 214 GB[2]. Besides, the complex mining process consumes a lot of energy. Hence, it is not cost-effective for a single peer to mine new blocks.

Therefore, S. Nakamoto [Nak08] already distinguished between full network nodes and leightweight nodes. The first hold a complete copy of the blockchain. The latter only possess a local copy of the headers. They synchronize their information with other network nodes, to ensure that they keep track of the longest chain. In practice, there are multiple different implementations of the Bitcoin protocol that allow variants of nodes, like a solo miner (that does not have a wallet) or a lightweight stratum wallet (that does not hold any information about the blockchain and does not participate in the mining process) [Ant14].

## 2.2 Public vs. Permissioned Blockchains

Bitcoin and Ethereum are designed as public blockchains. This means, that everyone can join the network using a software (peer) that runs a compatible protocol. In Bitcoin, everyone can operate as full peer, and therefore participate in the mining process, and in the validation and propagation of transactions. A full peer can also issue new transactions and holds a full replicated copy of the ledger. Hence, the contained data can be read by anyone in the public network.

In an enterprise environment read and write access is restricted to a predefined set of participants. Permissioned blockchains are designed for this purpose. A permissioned blockchain needs a central authority for identity and rights management. Peers create their keypairs on behalf of certificates issued by such authority services. Sophisticated privacy protocols on top of the blockchain consensus ensure data confidentiality.

## 2.3 Data Structures

The Bitcoin protocol [Nak08] defines the data structure of a blockchain. It is basically a linked list, where each list element corresponds to a block. Each block consists of a header section and a body. The body contains collected transactions, which represent atomic units of data (see Section 2.5). The header describes additional metadata used for mining and verification. The block-based data structure forms a ledger (see Figure 2.2). A new block is linked to the previous block by a calculated hash, that is included in the new blocks header and built upon the data of the previous block. The first block in a blockchain is called genesis block. Since it has no previous block, the according hash in the header section contains a null value. In Bitcoin, the genesis block is hard-coded.

[2] `https://www.blockchain.com/de/charts/blocks-size` Accessed: 27 April 2019

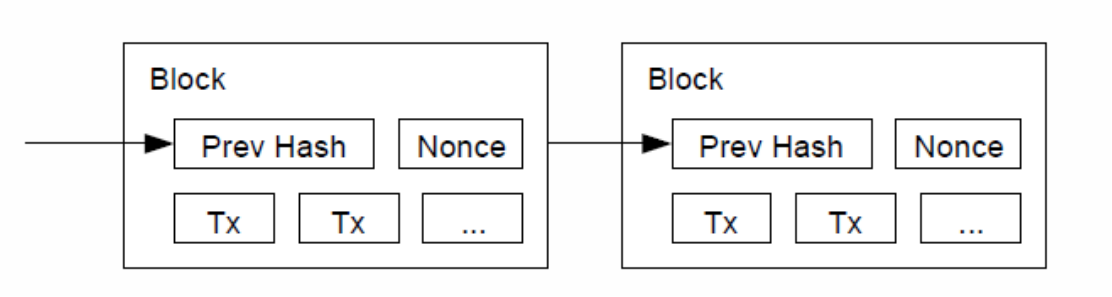


Figure 2.2: The blocks of the ledger are linked by hashes, building a "chain of blocks". Each block contains multiple transactions (Tx). This figure is a reprint from [Nak08].

A hash function takes one input of variable length and calculates a hash value of fixed length. It is computationally infeasible to find any two inputs that result in the same hash value [MVO96]. This ensures that any data modification is exposed by recalculation and comparison of the hashes in the blockchain. The blockchain is denoted as **immutable**, since the ledger keeps a record of all blocks, that cannot be changed by anyone.

Various other information is recorded in the block's header. The block header in Bitcoin for example consists of block version, merkle tree root hash, timestamp, nBits, nonce, and the parent block hash [ZXD+17]. The block version refers to the Bitcoin protocol. Nonce and nBits are used in the consensus mechanism (see Section 2.6). The timestamp is used for verification. It proves that the data existed at the time of block creation. The merkle tree proposed by Ralph C. Merkle [Mer80] is a binary hash tree, that consists of recursively calculated hash values. Leaf nodes form hashes of files. Another hash is built around the two file hashes. This is done recursively with all hashes until a single root hash is calculated. The merkle tree root hash in Bitcoin [Nak08] is used to save disk space. The protocol allows to discard transactions, after a sufficient number of blocks is appended to the block that contains the transactions. To keep the immutable characteristic of the ledger, the hash of the previous block is built upon the merkle tree root hash instead.

The peers in the Bitcoin network propagate and validate blocks. A valid block is included as head of the local blockchain copy. The Nakamoto consensus is based on the longest chain rule, meaning that the longest chain of blocks represents the correct state. When a peer receives a block, that is ahead of its local blockchain copy, it pulls the missing blocks from other peers in the network and starts its mining process on the new head. This guarantees that the local copy is always replaced by the longest valid chain. The linear chain of blocks in the original Bitcoin protocol leads to the creation of many stale (wasted) blocks and to unacceptable confirmation latencies. This problem is addressed by moving from linear arrangements to full directed acyclic graphs (DAGs).

The DAG is another data structure used in DLTs. It is applied by IOTA[3] blockchain,

[3] https://www.iota.org

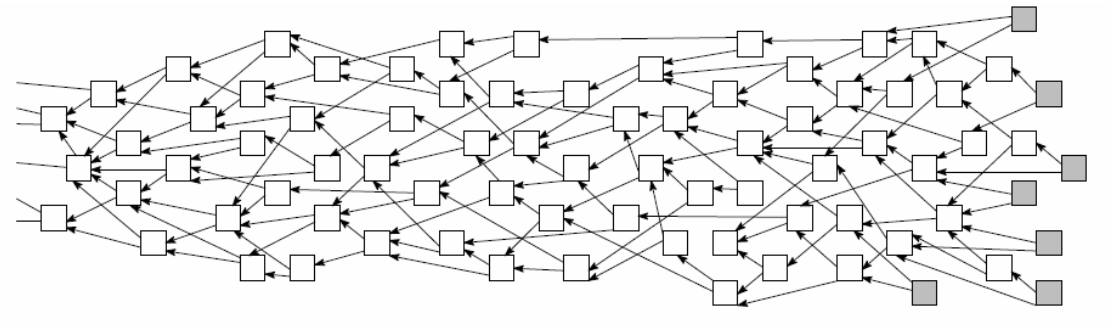

Figure 2.3: The IOTA-Tangle on high load. White squares represent valid transactions. Gray squares represent transactions waiting for approval, called tips. This figure is a reprint from [Pop18].

where it is introduced as the IOTA-Tangle [Pop18]. In a ledger, a split of the blockchain is possible if two concurrent blocks are propagated in the network at the same time. According to the Nakamoto consensus in Bitcoin, the longest chain will be selected for further mining. In such cases the discarded blocks of the shorter chain are called orphan blocks. Due to the mining complexity, the conflict is usually resolved after a few blocks. In a DAG, transactions from orphan blocks, that do not conflict with other transactions in the main chain are accepted as valid [XWS+17]. In a tangle, participants must constantly work to approve transactions in order to publish new transactions, i.e., each new transaction needs to approve exactly two previous transactions. Only non-conflicting transactions will be approved by others. Figure 2.3 shows an example of a tangle, where approvals are represented by directed edges.

## 2.4 Cryptography

Every participant in a blockchain network generates and holds one (or more) keypair(s). A keypair includes a public key and a private key. Identities in a blockchain are usually represented by addresses, which correspond to public keys. The public key is revealed to the other participants, while the private key must remain secret. A keypair can be used for data encryption and digital signatures. In blockchain networks, the first can be used to restrict data access to specific participants. The latter is used for proving ownership and to perform verification.

Data encryption can ensure **privacy** of a message sent between participants in a blockchain network. A secret message can be encrypted using the public key of a recipient. The private key is needed in order to decrypt the message. Hence, the only one who can decrypt the secret is the recipient. To create a digital signature, a sender signs the message using its private key. Everyone in the network, that knows the public key of the sender, can verify this signature using the sender's public key (**public verifiability**).

## 2.5 Transactions

The blockchain ledger maintains the state of a system at a certain time. A transaction represents a state transition. Transactions are created and published by blockchain peers. In Bitcoin, each participant holds a wallet, which has a balance of fungible assets (e.g., coins) and one or more assigned addresses. A transaction is used to transfer a specified amount of Bitcoin from the sender's wallet to a receiver wallet.

Bitcoin transactions contain a list of inputs and outputs and a digital signature. Every input of a transaction corresponds to an Unspent Transaction Output (UTXO) of a previous transaction. Each transaction has at most two outputs. One, to transfer the coins of the input(s) to another address (mandatory), and the other to return the change (optional). The balance of a wallet in Bitcoin is actually an aggregated value of UTXOs. A transaction can have one or more inputs. In Bitcoin a transaction is only valid, if the UTXO used as input, has not been used in an output of another transaction, and if the sum of all output coins is equal or less than the sum of all input coins.

Inputs and outputs allow to split and combine coins. Each output holds the amount of coins to be transferred, and a locking script, that specifies certain conditions that must be fulfilled in order to spent the output. In case of Pay-to-Public-Key (P2PK) the script contains the receiver's public key (see Section 2.7 for more details). As the sender signs the transaction with her/his private key, every peer in the network can validate the signature using the sender's known public key. Figure 2.4 shows transaction validation and signing. A valid signature captures the undeniable intention of the sender to transfer the respective coins to the receiver (**non-repudiation**).

## 2.6 Consensus

The consensus algorithm is a central building block of the blockchain technology. It is used to establish **trust** between the participants in a trust-less network. Within a P2P blockchain network the actual state is replicated. Every participant holds its own local copy of the blockchain. To prevent working on different versions of this data, the participants agree on a global valid state, which represents the "truth". This agreement process is called consensus.

Before Bitcoin, there were different approaches to implement a cryptocurrency. B-money introduced by Wei Dai in 1998 [Dai98] is cited in Nakomotos Bitcoin whitepaper [Nak08]. The protocols outlined in [Dai98] already use public key cryptography to provide pseudonyms and digital signatures and a computational puzzle to create new coins. This computational puzzle was introduced by Adam Back in 1997 [Bac02] as PoW algorithm called Hashcash. Hashcash was designed to prevent spam emails by attaching stamps to an email, that are created in a computational costly PoW algorithm.

The Bitcoin protocol [Nak08] includes a similar PoW algorithm as part of the consensus mechanism. A Bitcoin peer that participates in this consensus mechanism is called a

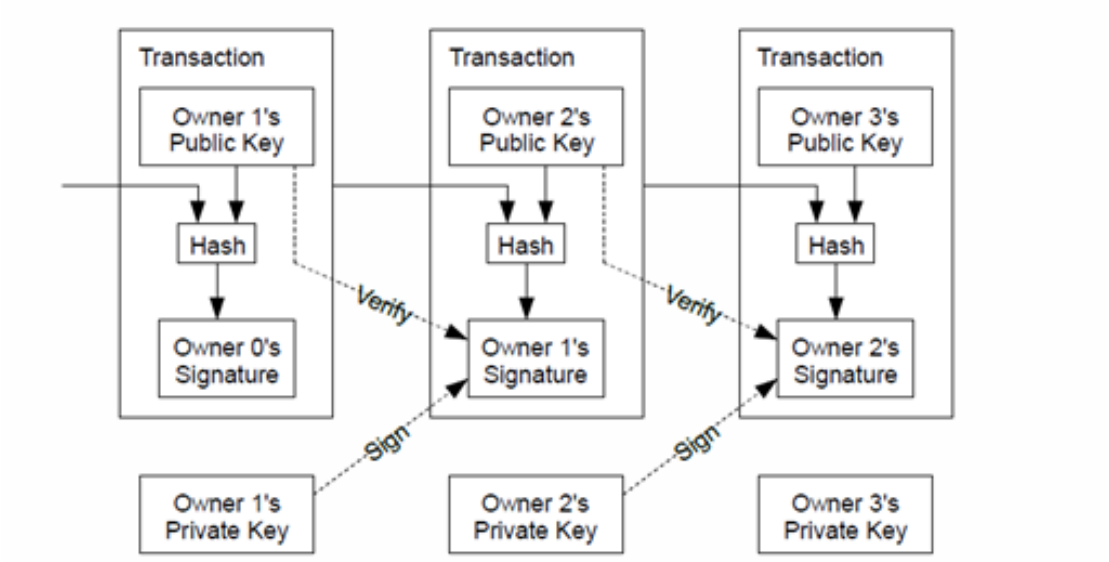


Figure 2.4: This chain of digital signatures represents a coin, passing from "Owner 0" to "Owner 1" to "Owner 2" to "Owner 3". Notice, that each transaction contains a link to the previous transaction and the new owners public key. This figure is a reprint from [Nak08].

miner. The motivation for the introduction of the PoW mechanism is to guarantee that no single participant (miner) is able to control the blockchain. Therefore, each miner creates a new block of pending (=uncommitted) transactions. A coinbase transaction creates new coins and allows the miner to collect transaction fees. It is packed into the block with the other transactions. The miner includes the hash of the last block in the blockchain into the block header. Then, the miner tries to solve a computational puzzle for this block. The puzzle can be described as follows: Try to find a SHA-256 hash with a predefined number of leading zeros, by adding additional arbitrary data to the block. This additional data is usually included in the nonce field of the blockchain header. The number of leading zeros is captured in the nBits field. Adding a leading zero increases the difficulty of finding a solution. In Bitcoin every 10 minutes a new block is created. The difficulty needs to be adjusted to the increasing number of miners, to maintain a block creation time of 10 minutes.

Once the puzzle is solved the miner publishes the block to the network. Each peer verifies the block by validating the data and recalculating the hash. If the block is valid it will be included in the local copy of the blockchain. The process of receiving and including a valid block stops the current mining process. The miner then restarts it with building a new block, that includes the hash of the received block in the header. If two blocks are published apparently simultaneously (i.e., due to network delays or network disconnections), a temporary split of the blockchain is possible. This is called a blockchain fork. In [Ant14] a detailed example of a forking event is illustrated. A fork

happens if a part of the peers receives one block and the other part receives the other block. They both include the block in the local copy of the blockchain and start their mining process, based on the hash of the received block. Bitcoin resolves this issue by considering the longest blockchain to be the correct one. Therefore peers discard blocks (orphan blocks), if they receive a block from a longer chain, and pull the intermediate blocks on demand. The probability of forks is decreased by the block creation interval of 10 minutes. But this interval and the restricted block size in Bitcoin cause a limitation in transaction throughput. That is one of the main drawbacks in Bitcoin.

The PoW algorithm is inefficient, slow and consumes a lot of energy. Since all of the miners try to find a solution for the same puzzle and this solution does not have any other value, it is impractical and unprofitable to use in an enterprise environment. Permissioned blockchains are mainly used in enterprise environments. They (a) establish trust a-priori between participants and (b) pre-define the participants quorum. This obviates one of the motives for PoW. Consensus can be reached by quorum based policies like majority votes. The problem of finding a reliable consensus in an untrusted distributed system is a well-known problem in computer science. Already in 1982, Lamport et al. proposed an algorithm for solving the byzantine generals problem [LSP82]. This is the basis for consensus algorithms used in Hyperledger Fabric or Tendermint. The systems and their consensus algorithms are described in Chapter 4.

## 2.7 Contracts

In Bitcoin the validation of a transaction is executed by a stack-based scripting language called Script. There are two types of scripts that must be executed on validation of a transaction: locking script and unlocking script. Bitcoin scripts contain basic instructions. Locking scripts are located in a transaction output and specify conditions for the spending of this output. Unlocking scripts satisfy the conditions and must be provided in order to spend the output. The most commonly used transaction script in Bitcoin is the Pay-to-Public-Key-Hash (P2PKH) script, that includes the public key hash of the receiver. If the receiver wants to use this output as an input for another transaction (spend the coins), she/he must provide an unlocking script that contains the receiver's public key and signature. The simpler form of the script is the Pay-to-Public-Key (P2PK) script, where the public key is used instead of its hash. Other Bitcoin transaction scripts are multi-signature, data output (OP_RETURN) and Pay-to-Script-Hash (P2SH). A detailed description of these scripts is given in [Ant14]. The Script language is not Turing-complete and therefore limited in its complexity.

When talking about contracts in context of blockchains, Ethereum's smart contracts are usually meant. Ethereum is besides Bitcoin the most popular public blockchain. It introduced smart contracts, which enable business logic on the distributed ledger. There are two kinds of addresses in Ethereum. The standard address (public key) used to identify participants, and a contract address for each instance of a smart contract. Therefore each contract has its own Ether (currency in the Ethereum network) balance.

A smart contract is a program that outlines an agreement between several parties and that automatically supports these parties in carrying out the associated processes. These programms are executed by each peer in a local sandbox environment, the Ethereum Virtual Machine (EVM). Once deployed on the blockchain, the smart contract cannot be changed by anyone. Each execution of a smart contract consumes a predefined amount of gas. Gas is a unit that specifies the computational effort for operations. The gas price represents "the fee the sender pays per computational step" [But14]. By setting a gas limit on contract deployment, a fail over mechanism (e.g., for infinite loops) is included in the contract execution. Smart contracts are written in script-like programming languages (e.g., Solidity, Serpent, Vyper) specifically designed for Ethereum. In contrast to Bitcoin's Script language, they are Turing-complete.

In permissioned blockchain systems, the concept of a contract is often understood differently. For example, Hyperledger Fabric sometimes refers to contracts as chaincode or smart contract, while Multichain uses smart filters to describe validation rules (see Chapter 4).

## 2.8 Summing-up: Fundamental Blockchain Properties

In the previous sections of this chapter intrinsic properties of the blockchain and the underlying technology principles are highlighted. These properties represent the main motive behind the huge interest in blockchain development. This section summarizes the properties that are outlined in the corresponding literature [XWS+17] [WG18] [ZXD+17].

According to [XWS+17], five fundamental properties of the blockchain technology can be identified:

**Immutability** After a block is verified and included into the blockchain by all peers its data is immutable. Malicious manipulation of the data is easy to detect by any peer in the network. This property is secured by the hash, that is calculated on each data block and included in the header of the next block.

**Non-repudiation** The immutability of the ledger combined with preceding and sophisticated verification and consensus processes provides non-repudiation.

**Integrity** The integrity of data is supported by cryptographic algorithms. Consensus mechanisms ensure that the replicated ledger stays consistent in the distributed peer network.

**Transparency** In a public blockchain everyone sees all transactions at any time (full transparency). In a private blockchain network at least the entitled parties of private transactions see respective data (partly transparent).

**Equal rights** Consensus mechanisms ensure trust between the participants through fair selection of block proposers and validators. The underlying peer-to-peer network is also based on democratic principles.

Besides integrity and transparency, Wüst and Gervais [WG18] describe the following additional properties:

**Public Verifiability** Every participant in a blockchain network is able to verify the ledger content and pending (=uncommitted) transactions. In public blockchains like Bitcoin, each full peer holds a full copy of the data, hence is able to verify the whole system state.

**Privacy** This property is not achieved in all blockchain systems, due to the distributed and transparent character. Anyhow, some of them provide data encryption (e.g., Hyperledger Sawtooth) or private channels (e.g., Hyperledger Fabric) to ensure data privacy.

**Redundancy** Redundancy is provided by data replication across multiple peers in the blockchain network. Centralized systems need backup storages to prevent data loss in case of a system failure.

**Trust Anchor** In a blockchain system, trust is ensured by majority votings on the correct state of the replicated ledger. Discrepancies are eliminated using consensus mechanisms.

Also, an additional important key characteristic that is not mentioned above is described in [ZXD+17]:

**Decentralization** The first public blockchain implementations were designed as fully decentralized systems. Permissioned blockchains used in business environments have at least a central identity management, but still manage the ledger state in a distributed manner.

CHAPTER 3

# Related Work

The cryptocurrency boom of the last few years created an enormous hype around the underlying blockchain technology. Companies seek for innovative use cases to improve their business processes. The idea of a secure distributed system inspires the development of new services for their customers. This leads to a variety of new blockchain-based applications. Different applications across all industries are being investigated by researchers in order to provide an overview and to show the technologies potential. The increased interest in the technology also led to a variety of new blockchain systems within a short period of time. Such rapid development results in a mixture of concepts and terminology. Furthermore, the simultaneous development of similar applications leads to large amounts of duplicated code. Therefore, a precise classification according to blockchain characteristics and the definition of reusable patterns are important research areas. This chapter shall provide an overview around current research activities addressing the mentioned challenges. Therefore, the first section of this chapter describes different classification systems for blockchains. Then, an outline of ongoing research around typical blockchain use cases and abstract patterns is given.

## 3.1 Taxonomy

In [ZXD+17], Zheng et al. outline properties to differ between the three main categories of blockchains: public blockchains, consortium blockchains and private blockchains. Those categories are originally described by Vitalik Buterin (the founder of Ethereum) in [But15]. While a public blockchain's mining process and ledger data is open to everyone, both consortium and private blockchain restrict access. A consortium blockchain is used for data replication between a set of industrial partners that need to collaborate. Therefore, the mining process is restricted to a predefined set of nodes. Private blockchains are used in one organization, that usually restricts access to the data and participation in the mining process. Zheng et al. also outline differences of these blockchain categories in

respect to efficiency, immutability, and centralization. Furthermore, according to their statement Zheng et al. compare the six most commonly used consensus mechanisms, namely Proof of Work, Proof of Stake, Practical Byzantine Fault Tolerance, Delegated Proof of Stake, Ripple, and Tendermint in respect to energy consumption, identity management of participants, and tolerated power of an potential adversary. Besides the different consensus mechanisms, Zheng et al. describe current challenges of blockchains (e.g., scalability, privacy leakage and selfish mining) and respective approaches to overcome those. Finally they identify possible future research directions.

Wüst and Gervais [WG18] use a similar basic classification approach as [ZXD+17]. They distinguish: permissionless blockchains, public permissioned blockchains and private permissioned blockchains. Furthermore, Wüst and Gervais use properties for blockchains in these categories to determine which kind of blockchain should be used or whether a blockchain should be considered at all. To classify the three categories, Wüst and Gervais differ between reader and writer. A writer is a peer that participates actively in the consensus process. A reader issues and analyses transactions, but will never extend the blockchain. Permissionless blockchains are comparable to public blockchains. A public permissioned blockchain restricts the set of writers but allows everyone to read the state. A private permissioned blockchain restricts both readers and writers. Wüst and Gervais developed the above mentioned recommendation system, that provides a flow chart. It contains questions about different requirements of the system (e.g., persistence of the state, the amount of writers). Each path in this chart represents a yes or no answer. Following the questions and paths throughout the chart one retrieves a recommendation for a suitable blockchain category.

In [XWS+17], Xu et al. propose a taxonomy for blockchains. They outline respective design decisions associated with decentralisation, storage and computation, configuration as well as design and deployment. First they discuss the concepts of a verifier and permissioning (similar to [ZXD+17] and [WG18]) and their impact on cost efficiency, performance, number of failure points and fundamental properties. A verifier is a trusted party that provides an external state. Xu et al. furthermore analyse the costs of on-chain and off-chain data storage and computation. They discuss configuration options for five design decisions: scope, data structure, consensus protocol, protocol configuration and new blockchain. In respect to the scope, they outline differences in the characteristics of public blockchains, consortium/community blockchains, and private blockchains. This classification is similar to [ZXD+17] and [WG18] and based on [But15]. The configuration options are summarized in a table and classified as less favorable, neutral or more favorable in terms of their fundamental properties, cost efficiency, performance and flexibility. Xu et al. then discuss anonymity aspects, incentives and deployment concerns. Finally, they propose a guide that shows how their outlined taxonomy can assist in the architectural design process of a blockchain-based system.

In this thesis a pattern-based approach is used to classify blockchains. The main emphasis is on the analysis of permissioned blockchains according to six blockchain interaction patterns, which are described in Section 5.2. After clarifying, that a permissioned

blockchain can be an appropriate solution for a specific use case (e.g., based on the recommendations of Wüest et al. [WG18]), one can apply the blockchain interaction patterns to find a suitable permissioned blockchain system. The taxonomies outlined in [ZXD+17] and [XWS+17] can be used in addition to examine the technical requirements for the blockchain solution.

## 3.2 Use Cases

In [CDP19], Casino et al. conduct a systematic literature review on scientific publications and grey literature in order to identify blockchain-based applications across different industries. Their qualitative analysis covers a total of 314 academic papers (e.g., peer-reviewed articles and journals, or conference proceedings papers) and 54 reports from a pool of grey literature, that were published up until April 2018. Casino et al. use an application-oriented classification approach. Based on their analysis, they identified nine main classes of applications, namely financial, integrity verification, governance, internet of things, health, education, privacy and security, business and industry and data management. Those classes are further divided into sub-classes. For example, business and industry includes applications used in the energy sector and for supply chain management. The sub-classes are explained in detail by outlining various publications in the respective field.

In [WG18], Wüst and Gervais analyse several use cases in order to show the applicability of their recommendation system (see Section 3.1). The use cases outlined by Wüst and Gervais are supply chain management, interbank and international payments, smart contracts, decentralized autonomous organizations, proof of ownership for intellectual property, e-voting, internet of things, land titles as well as trading and fair exchange protocols.

A comprehensive review on blockchain applications in the energy sector is given by Andoni et al. in [ARF+19]. Therefore, they reviewed 140 projects and startups to find potential benefits and drawbacks using blockchains in the energy sector. They discuss several applications in different application areas, including decentralized energy trading, green certificates and carbon trading, smart devices (IoT), grid management and e-mobility.

Golosova and Romanovs [GR18] analyse several blockchain applications to identify advantages and disadvantages of blockchains. They outline a variety of application areas similar to [CDP19], but on a smaller scale. Golosova and Romanovs describe blockchain-based platforms or applications for the government and the private management, the electronic voting, the authorship and the ownership, the goods and the raw materials, the diamonds, the medical, and the supply chain. They name for example MedRec [1], which is a prominent example of a blockchain application that provides transparent and secure access to medical records, or Blockverify [2], which provides an anti-counterfeit solution for different industries (pharmaceuticals, luxury items, diamonds and electronics).

[1] https://medrec.media.mit.edu
[2] http://www.blockverify.io

Similar to [CDP19], this thesis provides an application-oriented and therefore customer-oriented classification approach. The proposed method goes further, by outlining six community blockchain interaction patterns, that are universally applicable throughout a variety of application areas. A sample of the described blockchain applications from [CDP19], [WG18], [ARF+19] and [GR18] is used to evaluate this broad applicability in different industries (see Section 7.2).

## 3.3 Patterns

In [BP17], Bartoletti and Pompianu introduce often used programming patterns for Ethereum smart contracts. The motivation behind those patterns is to increase code quality and prevent vulnerabilities in smart contracts, since a contract cannot be changed once it is deployed on the blockchain. Therefore, Bartoletti and Pompianu describe nine common design patterns for Ethereum smart contracts. The patterns were derived by a manual code inspection of 811 smart contracts from the main Ethereum network. The source code of those contracts is published on the etherscan blockchain explorer[3]. The outlined patterns are Token, Authorization, Oracle, Randomness, Poll, Time constraint, Termination, Math and Fork check. Furthermore, the authors quantify the usage of the patterns by different categories: Financials, Notary, Game, Wallet and Library.

In [ET17], Eberhardt and Tai propose five off-chain design patterns based on their experience of developing blockchain applications for Ethereum. They describe solutions to outsource computation and data to an off-chain storage without comprising the main properties of the blockchain technology (e.g., immutability, transparency). The proposed patterns are: Challenge-Response (where instead of expensive on-chain checks, the final state is checked on the client side), Off-Chain Signatures (also known as payment channels), Content-Addressable Storage (store on-chain a reference to off-chain data), Delegated Computation (to outsource computation to an untrusted party), and Low Contract Footprint (to lower execution fees).

In [ZWSL17], Zhang et al. adopt four familiar software patterns (Abstract Factory, Flyweight, Proxy and Publisher-Subscriber) to address interoperability challenges in blockchain-based healthcare applications. They use an Ethereum-based self-developed healthcare app to show the applicability of the patterns. Especially healthcare applications require a high level of data privacy, since they deal with sensitive patient data. They face a challenge due to the large amount of patient data they process. The Abstract Factory pattern enables an instantiation of factory contracts, that manage evolving entities (realized as contracts) on behalf of the application. The Flyweight pattern is used to address challenges arising from the large amount of data. The Proxy pattern is applied in order to provide privacy by integration of an oracle. The Publisher-Subscriber pattern provides a solution for scalability problems.

[3] `https://etherscan.io/contractsVerified`

Other general design patterns for Ethereum smart contracts are outlined by Wöhrer and Zdun in [WZ18]. Similar to the work of Bartoletti and Pompianu [BP17], the authors identify patterns by researching common appearances of them in the Ethereum ecosystem. Instead of manual code inspection they use a Multivocal Literature Review (MLR) proposed by Garousi et al. [GFM16]. Starting with a systematic keyword search on white and gray literature, they created a pool of sources that are filtered in a following process to identify respective literature for Ethereum and Solidity smart contract patterns. Based on that literature (which included [BP17], [ZWSL17] and [ML17]), they applied an iterative pattern synthesis process to finalize their patterns. They are described in a pattern catalogue of 18 patterns, which are categorized in following classes: Action and Control, Authorization, Lifecycle, Maintenance, and Security. Analogue to [BP17] they identified an Oracle pattern and an Authorization pattern. Other application-based patterns described in this work are Pull payments (to realize withdraw functions), State machine (to realize control and execution flow with states), Commit and Reveal (to assure public verifiability on time-based enclosing of confidential data), Ownership (of a contract) and Access Restrictions (to check requirements for function execution).

In [LLX+18], Liu et al. outline eight design patterns for Ethereum-based (Solidity) smart contracts. They are based on their development experience and existing code base of smart contracts. They classify the patterns into four categories: Creational Patterns (Contract Factory, Contract Composer), Structural Patterns (Contract Decorator, Contract Facade), Inter-behavioral Patterns (Contract Mediator, Contract Observer) and Intra-behavioral Patterns (Hash Secret, Multi-signature). The authors investigate existing software design patterns for blockchain applications and conduct a case study, where they show how to apply the patterns to a real-world blockchain-based food traceability system called originChain[4]. The architecture design and contract interoperability is focused in this study.

Xu et al. [XPZ+18] propose a pattern collection of 15 design patterns for a variety of blockchain-based applications. The described patterns are based on implementations of real-world blockchain applications or derived from related literature on design patterns for blockchains. The patterns are assigned to four categories: patterns for interaction with the external world, data management patterns, security patterns and contract structural patterns. They include a Verifier pattern (= Oracle pattern), an Authorization pattern and a Tokenisation pattern similar to [BP17]. Furthermore, patterns for on-chain and off-chain storage, and state channels (to describe micropayments) are outlined. The collection includes a Factory Contract pattern similar to the Abstract Factory pattern described by Zhang et al. [ZWSL17].

Worley and Skjellum [WS19] integrate four patterns (Authorization, Oracle, Token, Poll) from [BP17] in their taxonomy. They provide a pattern catalogue with three abstraction layers, namely low-level patterns (Key-value Store, Address Mapping, Authorization), oracle patterns (Judge, Ticker Tape, Vote, Anti-Oracle, Blocklist, Announcement, Bulletin Board) and high-level patterns (Token, Migration).

[4] https://www.originchain.eu

The previous research on patterns is mainly based on existing code of Ethereum smart contracts and relevant literature. Since Ethereum is a public blockchain and smart contracts are written in specific programming languages (e.g., Solidity), the patterns identified in [BP17], [ET17], [WS19], [WZ18] and [XPZ+18] are strongly influenced by the design of Ethereum smart contracts. In this thesis, the outlined blockchain interaction patterns are derived from community blockchain use cases. These use cases are characterised by strong collaboration between individuals of a small group. In Section 5.2, six community blockchain interaction patterns are described, namely Transfer (P1), Exchange (P2), Sharing (P3), Chat (P4), Election (P5) and Tracking (P6). There is a relation between the outlined community blockchain interaction patterns and the patterns described by [BP17], [ET17], [WS19], [WZ18] and [XPZ+18]. This is reasoned by the way the patterns are derived. While in this thesis, typical community blockchain use cases are examined to find possible abstractions, the patterns outlined in those papers are derived from manual code inspection, literature research and development experience. The concepts described by the outlined community blockchain interaction patterns should ideally complement and interlock with current pattern research. Some of the patterns from the given literature describe possible technical solutions for a specific requirement in the outlined community blockchain interaction patterns.

The Oracle pattern in [BP17], the Ticker Tape pattern in [WS19], the Oracle (data provider) pattern in [WZ18], and the Verifier pattern in [XPZ+18] are closely related to each other. They are represented as part of the Tracking pattern (P6) in this work. The Delegated Computation pattern described by Eberhardt and Tai in [ET17], includes zero-knowledge proofs, that represent a possible solution for the linkability problem in the Election pattern (P5). The On-Chain Encryption pattern and Off-Chain Data pattern in [XPZ+18] enable data privacy, which is needed in the Election pattern (P5) and for private 1:1 communication in the Chat pattern (P4). The Token pattern from [BP17] (which is integrated in the pattern catalogue in [WS19]) and the similar Tokenization pattern from [XPZ+18] describe fungible objects (tokens) that are needed to realize the Transfer pattern (P1), the Exchange pattern (P2) and the Sharing pattern (P3). The Time constraint pattern in [BP17] is based on contracts in Ethereum. However, time-triggered events are an important requirement in the Sharing pattern (P3). The Election pattern (P5) is similar to the poll pattern from [BP17], but adds more requirements to the pattern, which are based on real-world elections. The Key-value Store pattern outlined by Worley and Skjellum in [WS19] is used to store and retrieve data effectively, which is a precondition to the Tracking pattern (P6). The Announcement pattern in [WS19] describes a contract for announcements from authorized contracts. This pattern is based on Ethereum smart contracts. The idea for an autorized set of writers and the retrieval of data by other contracts is similar to the concept of the Tracking pattern (P6). But the Tracking pattern (P6) includes efficient data retrieval as important requirement. The Bulletin Board pattern in [WS19] describes an application of the Chat pattern (P4) similar to the Q&A-Board use case (see Section 5.1.3). The Authorization pattern in [BP17] (which is integrated in the pattern catalogue in [WS19]) describes a permissioning system, where function execution is restricted to a predefined address.

In [WZ18], Wöhrer and Zdun define two Authorization patterns, namely Ownership and Access Restriction. While the Ownership pattern restricts the function execution to the owner (creator) of a contract, the Access Restriction pattern allows to define a (set of) condition(s) that must be met in order to execute a function. The Access Restriction pattern is similar to the Embedded Permission pattern in [XPZ+18]. The pattern collection in [XPZ+18] additionally contains the Multiple Authorization pattern. It describes a mechanism to authorise transactions by multiple addresses (participants). This can be used for multi-signature transactions. Authorization is an important part of all community blockchain interaction patterns (P1-P6). Ownership of fungible and non-fungible objects is a central building block for the Trading patterns (P1-P3). Access restriction is important, when using different user roles like an Expert role in a Q&A-Board (see Section 5.1.3). Therefore, it is part of the Chat pattern (P4). In the Election pattern (P5) only authorized participants are allowed to vote. The Tracking pattern (P6) allows an authorized oracle service or participant to add certified data into the blockchain.

CHAPTER 4

# Permissioned Blockchain Systems

Due to the huge interest in the blockchain technology, the amount of enterprise blockchain solutions increased tremendously in the last few years. This is justified by the benefits and inherent properties of a blockchain. An important difference between Cryptocurrencies (e.g., Bitcoin) and permissioned blockchains is the restriction of network access. While in a public blockchain anyone in the world can join the network and participate in the consensus mechanism, a permissioned blockchain restricts access to authorized participants only. Therefore, companies prefer applications based on permissioned blockchains.

Permissioned blockchains in general enforce modularity by introducing components for identity management, role based access control, data encryption and decryption, transaction management and consensus. This chapter provides an overview of selected permissioned blockchain systems that are developed by big companies like IBM, Intel, or JPMorgan Chase & Co. They provide different consensus protocols and components. The main differences between the systems are outlined by analysing the basic concepts of blockchains mentioned in Chapter 2, namely Permissions, Transactions, Consensus and Contracts. Furthermore, their individual solution components are described in this chapter. All information is based on literature reviews of official whitepapers or developers documentation.

## 4.1 Corda

The Corda Platform[1] was initially introduced 2016 in a whitepaper [BCGH16] published by Richard Gendal Brown, James Carlyle, Ian Grigg and Mike Hearn from R3. It is designed to establish interoperability of financial applications on a distributed ledger solution. Details about Corda's components are outlined in the successor [Hea16] of the original whitepaper and the accompanying technical whitepaper [Bro18].

[1] https://www.corda.net

**Solution components** Corda is based on three components: application logic (smart contracts), notary pools and a flow framework. The application logic is described in deterministic smart contract code, which can be written in Java or Kotlin[2]. Corda's smart contracts represent constraints to check validity of state changing transactions, according to a pre-defined rule-set. Notary pools are used to order transactions in respect to their timestamps and resolve conflicts by eliminating duplicates of the same transaction. The flow framework supports users to create complex multi-step protocols for collaboration of mainly distrusting parties.

**Contracts** In Corda (parts of) real-world contracts are represented by a state object. Corda differentiates between contracts and the actual state of a contract, which holds an instance, version and status of this particular contract. Consensus between the parties is based on the states. Transactions trigger updates on state objects by consuming old state objects and creating new ones.

**Transactions** Corda's transaction concept is based on a UTXO model (like it is used in Bitcoin). A transaction contains zero or more input references (= state objects to be consumed). Unlike in Bitcoin, the transaction outputs represent rather complex state objects than simple values. Each transaction contains zero or more output states (= newly created state objects). On creation, each transaction receives a unique identifier, the transaction ID. When a state object is created in a transaction, it is associated to the transaction ID and the state's index in the output references.

Besides input references and output states, a transaction contains multiple other fields: Attachments, Commands, Type, Timestamp, Signatures and Summaries. In Corda each contract can access zip files that contain additional data, certificates or code for the transaction. The zip file hashes are stored in the Attachments field of a transaction. A command is used to distinguish the tasks that could be performed on an asset. A contract specifies certain verification routines for inputs and outputs based on these commands. Because different tasks can involve different participants, each command requires to specify a list of public keys (in Corda these are actually composite keys), which have to sign the transaction. For transactions with inputs having varying notary states, the notary-change type can be set explicitly. This will re-point the input states to the same notary.[3] The Timestamp is optional, and represents a time window for the occurrence of the transaction. Each transaction needs to be signed by the public keys of the defined command. Summaries describe the purpose and tasks of a transaction in natural language (English). Contracts check it against the actual content of the transactions to find possible discrepancies.

**Consensus** The Corda consensus mechanism is based on two building blocks, namely transaction validity and transaction uniqueness. The former ensures that the contract

[2] https://docs.corda.net/head/key-concepts-contracts.html

[3] http://docs.corda.net/releases/release-M9.2/key-concepts-consensus-notaries.html

code associated to the updated state is valid, in respect to determinism. A transaction is valid if it contains all the signatures and all dependent transactions are also valid. The latter ensures that all of the input states must not been consumed by any other transaction.

In a first step, transaction validity is checked on every peer separately. Therefore, the same valid transaction appears multiple times in the network. The elimination of duplicates of a transaction is performed by notary pools in a second step. There can be many notary pools, which are formed by distrusting participants. Only parties with interest in a specific transaction are allowed to see the corresponding state. This is predefined in the contract code.

**Permissions** Permissioning in Corda is based on identity management. Each participant needs an identity that has an associated valid certificate, in order to join the network. This certificate is issued by a trusted root authority. Corda identities can be individuals or organizations. They are represented by X.500 names. Only one public key per identity is allowed. This is an important feature for financial applications, where identities represent legal entities. It guarantees that actions, that require a signature (e.g., issuance of a transaction), are legally binding and undeniably connected to a legal entity. By design, Corda identities do not necessarily need to be legal entities, but they need to be unique.

## 4.2 Hyperledger Fabric

The Hyperledger Project[4] was founded in 2015 by the Linux Foundation. It combines individual open source blockchain infrastructure projects like Fabric, Burrow, and Sawtooth in a cross-industry collaboration. The main rationale behind Hyperledger Fabric[5] is to provide a modular architecture for deployment of a blockchain in enterprise environments [Fab19].

**Solution Components** As Hyperledger Fabric is a permissioned blockchain, it needs additional components, such as Membership Service Providers (MSPs) and Ordering Services, to establish trust between participants. Participants need to enroll via MSP to the blockchain network. To keep the blockchain modular, not only MSPs are exchangeable, but also the consensus mechanism.

Hyperledger Fabric maintains two components for data management: world state and transaction log. The world state represents the ledger state at a specified time. All of the transactions that resulted in a valid state transition previous to the current world state are kept in the transaction log. Both components build the blockchain ledger.

[4] https://www.hyperledger.org
[5] https://www.hyperledger.org/projects/fabric

**Contracts** Business logic is written in chain code (similar to Ethereum smart contracts). It is executed by applications, when changing the ledger state. Chain code is written in general-purpose programming languages. The officially released SDKs support Node.js and Java. SDKs for Python and Go are also available, but are not officially released and supported yet[6].

**Transactions** Hyperledger Fabric transactions contain general metadata (header) and signature, as well as other essential fields used for consensus: proposal, response and endorsements. A proposal provides input parameters, used when triggering the execution of chain code by an application. A response holds the values of the world state before and after chain code execution. Each transaction additionally holds a list of endorsements. This field is used in the second phase of the consensus mechanism.

**Consensus** Consensus in Hyplerleder Fabric is based on Practical Byzantine Fault Tolerance (PBFT). It is split into three phases: (1) Proposal phase, (2) Packaging phase, and (3) Validation phase. Peers can take different roles in the consensus process, namely the role of an endorsement peer or an orderer.

Phase (1): When a client wants to interact with another client, the application sends a request to the SDK. A transaction proposal is generated, which contains important input parameters for the requested chain code. According to which endorsement policy is chosen for the chain code, a certain amount of endorsing peers need to endorse the result of the transaction (= agreement). The endorsing peers return a proposal response (= read/write sets on world state + signature) after running respective chain code on their local machines. The returned endorsements are validated and compared by the SDK. If all responses return the same values and the SDK received enough endorsements, according to the endorsement policy, the transaction is passed to the ordering service (ordering peer).

Phase (2): When an ordering peer receives a transaction, it orders it chronologically in respect to other transactions. This is done by each channel. Then, it packages the transactions into blocks for each channel.

Phase (3): The blocks are forwarded to all peers in the channel. The peers verify, that the endorsement policy and read/write sets are correct for each encapsulated transaction. If the block is valid, it is inserted into the local copy of the blockchain. Furthermore, updates on the world state are performed. Additionally, applications receive events triggered by the execution of updates.

**Permissions** To preserve privacy between participants Hyperledger Fabric introduces the concept of channels. A channel is a separate ledger in which transactions, members and other informations are only visible to members of that channel.

[6] `https://hyperledger-fabric.readthedocs.io/en/release-1.4/getting_started.html#hyperledger-fabric-sdks`

## 4.3 Hyperledger Sawtooth

Hyperledger Sawtooth[7] is an open source blockchain platform. It was initially proposed in April 2016 by Mic Bowman (Intel) and Richard G. Brown (R3)[8]. Sawtooth can be used either as permissionless or permissioned blockchain (by setting different configuration properties). Similar to Hyperledger Fabric, the platform provides a modular architecture design including pluggable consensus algorithms. The main difference targeted by Sawtooth is that consensus can be changed on the fly on a running blockchain network. Potential application areas of Sawtooth are financials and IoT.

**Solution Components** Sawtooth consists of multiple components, such as the validator network, an event system, the global state (represented by a merkle-tree), the consensus engine, and a journal (responsible for block managment). It introduces a new concept of transaction families, which can be implemented by users to customize the blockchain functionality according to their needs. To support custom development a set of core transaction families is given (e.g., IntegerKeys, Settings, Identity), that can be used as examples.

**Contracts** Sawtooth is able to run Ethereum smart contract code written in Solidity using the Seth transaction family. It is also possible to write contracts as native business logic in different general-purpose programming languages. Respective SDKs support Python, JavaScript, Go, C++, Java and Rust. Even applications with different types of contracts are supported on the same blockchain network. This is enabled by customizing different transaction processors for each application.

**Transactions** In contrast to other blockchains, Sawtooth transactions are processed in packages called batches. If a state transition is performed, all transactions of a single batch are committed to the state at once. Batches are used to resolve cyclic dependencies of transactions. To link batches and transactions, each transaction header holds the public key of the batch creator. This prevents unauthorized repackaging of already used transactions.

Each transaction consists of a serialized transaction header, the issuer's signature and the payload, which represents the actual family-specific data (family-specific means that the data is interpreted according to a pre-implemented transaction family). Furthermore each transaction can have dependencies to preceding transactions. Therefore a list of such transactions can be specified in the header section. Input and output fields define state addresses from which the state is read or to which the new state is written by an application. More details about the transaction family (which is used to interpret the data) are included in the header section.

[7] https://www.hyperledger.org/projects/sawtooth
[8] https://wiki.hyperledger.org/display/sawtooth

Multiple batches are then included into a block. The Sawtooth journal consists of different components for batch and block validation, as well as creation and distribution of new blocks. These components communicate with a consensus interface, which determines whether a block proposer is allowed to publish a new block. It is also responsible for validation of published blocks in respect to predefined consensus rules.

**Consensus** Sawtooth introduces the Proof of Elapsed Time (PoET) consensus protocol. Besides it also supports a PoET simulator (to simulate PoET consensus on different hardware types and virtual cloud environments), and a simpler consensus based on random-leader selection (for testing and development only). RAFT [OO14] consensus protocol is currently under development [OBM+18].

The Sawtooth consensus, Proof of Elapsed Time (PoET), is a sophisticated random-lottery leader election. It is executed on Intel Software Guard Extensions (Intel SGX), which represent a trusted execution environment. The algorithm uses an enclave. An enclave is a secured function in the address space of an application. The enclave assigns each validator on a request a waiting time. Then the wait times are compared and the validator with the shortest time takes over the role of the leader. Only the leader is allowed to publish the next block.

**Permissions** The privacy model in Sawtooth covers access restriction for the validator network and for transaction or batch proposal.

## 4.4 Multichain

Multichain[9] was firstly introduced in a whitepaper [Gre15] published in 2015 by Gideon Greenspan, the CEO of Coin Sciences Ltd., which is a UK blockchain technology company. Well-known IT-consulting firms, like SAP and Accenture, are listed Multichain platform partners[10], that build blockchain solutions based on Multichain.

Multichain is derived as a fork from the official Bitcoin network client called Bitcoin Core. The interface supports a simple command-line and API. This makes it fully compatible to the original Bitcoin Core. Therefore each peer in the Multicain network can act as a peer on the original Bitcoin network by changing configuration properties.

**Permissions** Multichain extends the Bitcoin protocol with an additional handshake process, that enables permissioning on the network. This handshake is described in the Multichain whitepaper [Gre15] as follows:

> "1. Each node presents its identity as a public address on the permitted list.

[9] https://www.multichain.com
[10] https://www.multichain.com/platform-partners

2. Each node verifies that the other's address is on its own version of the permitted list.
3. Each node sends a challenge message to the other party.
4. Each node sends back a signature of the challenge message, proving their ownership of the private key corresponding to the public address they presented."[11]

A peer can abort the connection if the handshake fails.

The Multichain network is based on roles and permissions, that are extended to restrict the right to send or/and receive transactions, or the right to propose new blocks. The creator of the first block in the blockchain, a so-called genesis block inherits the role of an administrator. She/he is allowed to grant new permissions to other users. To keep democratic principles, Multichain introduces a voting mechanism for administrators in order to change user permissions. Conflicts are resolved by restricting new block proposals. They can only be issued by miners with higher or at least equal permissions in comparison to the permissions granted by transactions of the block.

Blockchain streams[12] are used in Multichain to restrict data visibility. This is ensured by encrypting data before including it to the blockchain. The process uses a mix of symmetric and asymmetric cryptographic algorithms.

**Transactions** Transactions in Multichain are similar to Bitcoin transactions. They include a set of inputs, a set of outputs and metadata (e.g., version, locktime). Besides an JSON-RPC API for the creation of basic transaction, Multichain allows to create raw transactions[13]. They provide more flexibility, for example in the selection process of UTXOs or enable complex permissioning.

**Contracts** Instead of contracts, Multichain provides smart filters[14], that represent customized validation rules for transactions and data. There are two types supported, namely Transaction filters and Stream filters. Transaction filters describe validation rules for transactions. They are checked against inputs, outputs and matadata of a transaction. Stream filters define validation rules for stream items. These rules set conditions on both on- and off-chain item data, as well as key and publisher of an item. A smart filter is written in JavaScript. To eliminate non-determinism certain functions (e.g., Date.now(), Math.random(), Math.sin()), the access to external serivces and multithreading are disabled.

[11] [Gre15] page 5.
[12] `https://www.multichain.com/developers/data-streams`
[13] `https://www.multichain.com/developers/raw-transactions`
[14] `https://www.multichain.com/developers/smart-filters`

**Consensus** Consensus in Multichain is a round-robin based protocol, where new blocks are created by permitted miners in rotation. Unlike multiple validators in Practical Byzantine Fault Tolerance (PBFT) algorithms, the Multichain consensus selects only one validator for each block. Blocks are valid if (1) permission changes are applied in the specified order, (2) and the miner of the block does not appear as miner in the (last spacing - 1) blocks, where spacing is the rounded result of the number of permitted miners resulting from the changes multiplied by the mining diversity.

An additional mining diversity parameter restricts the number of blocks created by a miner in a specified interval. The value is set between 0 and 1, where 0 stands for no restriction at all and 1 specifies that all of the permitted miners are included in the rotation. The suggested value of the mining diversity parameter is 0.75, as it ensures safety without blocking the mining process, which can happen in case of inactivity of participants. As within Bitcoin, the longest chain represents the true state of the blockchain.

Although Multichain supports the co-existence of multiple private blockchains, it does not support atomic swaps. Atomic swaps establish exchanging mechanisms between assets of different blockchains.

## 4.5 Quorum

Quorum[15] is an open source permissioned blockchain that was initially introduced in 2016 in a whitepaper [Quo16]. Quorum is based on the Ethereum blockchain. It adapts the protocol to support a blockchain that keeps track of both a public and a private state, maintained by each peer. The private state holds encrypted private transactions that restrict read permissions to only a few participants. The Quorum protocol includes modifications of transactions and consensus principles. Besides that, Quorum uses programmable smart contracts that run on the EVM.

**Permissions** Every participant holds one consented public state and several private states, both supported by the same blockchain. While public transactions are validated by all participants, private transactions are validated only by the respective entitled parties of the transaction. The validation process is conducted by execution of the associated smart contracts. This is done on each peer running the EVM. Public and private states are stored in two separate patricia-merkle trees.

**Solution Components** The components establishing more privacy on the blockchain are the Transaction Manager, Network Manager, Crypto Enclave and the Quorum Chain. The Transaction Manager establishes communication to other Transaction Managers to send and receive transactions. It provides access of private transactions to encrypted data and the local data store on each peer. The Crypto Enclave handles encryption

[15] `https://www.goquorum.com`

and decryption for private transactions and is responsible for the key management. The Network Manager restricts the network access to a predefined set of participants (permissioned peers). The Quorum Chain represents the enhanced consensus mechanism.

**Consensus** In Quorum consensus is built upon a subset of peers. Such peers are allowed to vote on blocks, to determine the head of the blockchain at a current height. Quorum implements both RAFT [OO14] and Istanbul BFT consensus[16]. Only peers with a marker role are allowed to propose new blocks. Quorums consensus mechanism is implemented as a smart contract. It contains marker and voter lists and describes further details to manage the network.

**Transactions** A Quorum transaction can be identified as private by setting an optional list of entitled parties and an additional field containing the hash of the encrypted private data. Other than that it is structured like a standard Ethereum transaction, although it does not require an Ether balance. Private transactions create private contracts.

**Contracts** Since, Quorum is based on the Ethereum blockchain protocol, it uses smart contracts to describe business logic. The contracts are written in standard Solidity. In contrast to Ethereum smart contracts, they can be defined as public or private. Private contracts restrict visibility and execution to a predefined set of participants in the network. There are two restrictions on private smart contracts: (1) they cannot update public contracts and (2) once deployed as public contracts, they cannot be made private later on.

## 4.6 Tendermint

Tendermint[17] was initially introduced by Ethan Buchman in 2016 [Buc16]. Unlike in Bitcoin, Tendermint strictly separates consensus mechanism and the P2P network from the application logic and state. While transaction validation, maintenance and querying of uncommitted transactions are performed on the application layer, the immutable order of the transactions and the propagation of blocks and transactions is ensured by the Tendermint Core component.

**Solution Components** Tendermint's two main technical components are the ABCI (Application BlockChain Interface) and the Tendermint Core. The Tendermint Core component serves as blockchain consensus engine. It ensures the consistent order of transactions over all replications of the blockchain. Tendermint consensus is based on Byzantine Fault Tolerance (BFT). This ensures that the system works correctly on a 2/3 majority of nodes. Therefore consensus is reached even if 1/3 of the nodes fail or act

[16] `https://github.com/ethereum/EIPs/issues/650`
[17] `https://tendermint.com`

maliciously. The ABCI allows to build flexible applications on top of the Tendermint Core consensus engine.

**Contracts** The ABCI provides an interface for distributed applications to communicate with the underlying Core component via a Socket Protocol (TSP). It ensures that applications can be written in a general purpose programming language. There exist different implementations or libraries of the ABCI for Go, Java, Python, C++, Javascript, Rust, Erlang and Ocaml[18].

There are three ABCI connections for different purposes. The Consensus Connection maintains the block proposals and is responsible for the execution of blocks. Operations of this connection type read and update the state. The Mempool Connection ensures transaction validation of pending transactions. Such transactions are stored in a local mempool before they can be included into a block. Only valid pending transactions get relayed to other peers in the blockchain network. The Info Connection supports the querying of the application state and is used for initialization. Operations of Mempool Connections and Info Connections only read the state, but do not change it.

**Transactions** Transactions in Tendermint are byte arrays of arbitrary length. They do not have a predefined structure, like in other blockchain systems. Applications built on top of the Tendermint blockchain system can design their own transactions in order to fit their needs. Hence, the application is fully responsible for transaction validation. Tendermint provides two message types that are sent from the Tendermint Core (consensus engine) to the application, namely CheckTx and DeliverTx. The main difference between these messages is that they occur in different stages of the consensus mechanism. CheckTx is used for the validation of pending (uncommitted) transactions only. The application returns the result of the validation by sending a ResultTx message. DeliverTx is sent for each committed transaction. Committed transactions are included in the blockchain through successful consensus. They are again validated by the application. After validation, the application applies the encapsulated state transitions to the local state. Pending transactions are stored in the Tendermint mempool. Only valid transactions are forwarded to other peers in the network. Invalid transactions are discarded.

**Consensus** The Tendermint BFT is a round based approach. Every peer is called validator and in each round exactly one validator is designated to propose a block for the current height. The height corresponds to the number of blocks in the blockchain. All other validators vote whether or not they want to include this block into the chain. This is the case if a 2/3 majority of the validators agree on the block in the same round. Tendermint distinguishes between two phases of the validation process called the pre-vote and the pre-commit. First 2/3 of the validators need to pre-vote for the same block. This is called a polka. Then again 2/3 of the validators need to pre-commit for the same

[18] `https://tendermint.com/ecosystem`

block. A polka is the precondition for a successful pre-commit. If this process fails, the next round starts with a new validator acting as proposer of the next block.

**Permissions** Tendermint restricts the set of validators that are allowed to participate in the consensus mechanism. The set of permissioned validators can be defined in the genesis file. A new validator can be added to this list by sending an EndBlock message. All of the permissioned validators are expected to be always available online[19].

## 4.7 Other Blockchain Solutions

The list of new emerging permissioned blockchain solutions is increasing continuously. Since, this work is no attempt to provide a complete collection that explains all details of every existing permissioned blockchain system, only a few are outlined in the previous section. Still, there are other candidates that were examined, but not selected for the pattern-based analysis, conducted in Chapter 6. In this section, other blockchain solutions that have certain similarity with, or are built upon, the previously described systems are outlined briefly.

### 4.7.1 BigChainDB

BigChainDB[20] was founded in February 2016 as a decentralized database solution. It integrates Tendermint consensus and network mechanics to enable benefits of a real blockchain (e.g., owner-controlled assets, decentralisation, and immutability) while keeping basic database properties (e.g., low-latency, high capacity and throughput). Although BigChainDB describes its own transaction specification, replication and consens is performed over Tendermint network [Big18]. Therefore, it is arguable that BigChainDB is not a permissioned blockchain itself, but rather an application based on Tendermint. Typical application areas are supply chain management, IPRs, identity management, IoT and data governance.

### 4.7.2 Hyperledger Burrow

Hyperledger Burrow[21] was first released in December 2014 as Eris DB by Monax Industries. Since 2017, it is one of the frameworks contributed under the Hyperledger project. Burrow represents a permissioned blockchain client that implements the EVM to execute Ethereum smart contracts on a Tendermint consensus engine. Each client consists of three major components: the permissioned EVM, the Tendermint consensus engine and a RPC gateway. Burrow is implemented in Golang. Burrow supports smart contracts written in Solidity. The project is licensed under Apache 2.0 and is currently in incubation phase[22].

[19] https://www.tendermint.com/docs/tendermint-core/validators.html
[20] https://www.bigchaindb.com
[21] https://www.hyperledger.org/projects/hyperledger-burrow
[22] https://wiki.hyperledger.org/display/burrow

### 4.7.3 Hyperledger Iroha

Hyperledger Iroha[23] was initially proposed in September 2016 by Makoto Takemiya (Soramitsu), Toshiya Cho (Hitachi), Takahiro Inaba (NTT Data), and Mark Smargon (Colu) [24]. Iroha was intended as permissioned blockchain for the IoT. The main idea of Iroha is to provide simple commands and queries to enable fast application development on a permissioned blockchain. Iroha itself is written in C++ and supports programming of smart contracts in the same language, and also in Python, Java, Javascript. There are also SDKs available for iOS and Android [25] The newly introduced consensus in Iroha is called Yet Another Consensus (YAC) [MLI+18] and based on BFT. Additional to common networking permissions, Iroha supports a fine-grained permissioning system, where permissions can be set even on single commands and queries.

[23] https://www.hyperledger.org/projects/iroha
[24] https://wiki.hyperledger.org/display/iroha
[25] https://github.com/hyperledger/iroha#is-there-sdk-available

CHAPTER 5

# Developing Community Blockchain Patterns

Previous conducted studies to identify design patterns in blockchain-based applications, which are summarized in Section 3.3, are based on a source code inspection, development experience or literature study of existing smart contracts in the Ethereum blockchain. As a result, only applications that are currently implemented or proposed as Ethereum smart contracts are examined in this process. Therefore, a vast majority of the outlined patterns are custom-designed for Ethereum smart contracts and/or the properties of their programming languages (mainly Solidity). Since Ethereum is a public blockchain, the requirements of the deployed applications benefit from their inherent properties (i.e., by using the built-in cryptocurrency), but also need to deal with their limitations (i.e., using off-chain storage for scalability reasons or data encryption to pursue privacy). These properties influence the proposed patterns.

This thesis presents another approach to find applicable design patterns for the development of community blockchain applications. Since community processes require both a high degree of collaboration and high autonomy of the individual participants, these use cases benefit from inherent properties of distributed architectures. Therefore, in Section 5.1 typical community use cases are identified. Furthermore, use cases for a collaborative example application (a virtual media library) are described. Based on these use cases, common collaboration patterns are derived and proposed in Section 5.2 as community blockchain interaction patterns. Furthermore, the derivation of these patterns from the described community blockchain use cases is outlined in Section 5.3.

## 5.1 Use Cases for Community Blockchains

Community members need to collaborate on different topics or projects. This is done in working groups. A working group is a group of people (e.g., developers, experts), who

work together to pursue a specified goal. Such group-based processes are often managed in groupware, which is a tool for the technical support of communication, collaboration and coordination of groups and their processes [Ell91]. Depending on the aim of the working group (e.g., participation in workshops/events/expert talks, or the acquisition and shared use of expensive resources), different privacy aspects need to be considered. Transparency is a main goal of collaborative processes, as it enhances participation and trust among the working group members (that do not necessarily know each other). The collaborative nature of working groups is also supported by democratic decision-making processes.

In Chapter 2, the inherent benefits of the blockchain technology were outlined, including public verifiability, transparency, privacy, integrity, redundancy, and trust as described by [WG18]. Since the requirements for cooperation of members in working groups target those principles, this section includes a collection of collaborative group-based business processes.

### 5.1.1 UC1: Acquiring and Sharing Resources

**Motivation** High sophisticated technical equipment (e.g., 3D Printer) is in general very expensive. Expenses include not only the acquisition costs, but also costs for maintenance and operation. For resources, which are not used 24/7 it is much more efficient for individuals to share costs and usage.

**Description** A group of individuals would like to buy a resource and share the usage accordingly. In a first step, each party transfers its share of the acquisition costs to a joint account. Then, using this balance the resource is bought from a seller. The joint account is used further for paying the running costs. Each individual acquires the right to use the resource. To reduce conflicts the parties agree on a time schedule for a fair shared usage.

**Actors**

- Buyer - a group of individuals, who share the same interest in a resource.
- Seller - someone who offers and sells the resource and has ownership.

**Preconditions**

- There must be an offer for the resource, including price or other conditions.
- The seller must be the owner of the resource.
- Ownership must be transferable to a group of individuals.

- Each individual of the group has an account with a positive balance of fungible objects.
- The seller has an account to receive fungible objects.
- There is a joint account, that is managed by a contract.

**Postconditions**

- The group owns the resource.
- Usage right is held by each individual of the group.
- The seller's account balance is increased by the price of the resource.
- In case of a failure (e.g., one of the individuals did not pay enough), the correct amount of fungible objects is returned to each sender.

### 5.1.2 UC2: Organizing Meetings with Automated Room Reservation

**Motivation** In a working group or other communities face-to-face meetings are necessary from time to time. There are also other occasions like workshops or team building events where a personal meeting is preferable. People nowadays use software (e.g., Doodle[1]) to agree on time. This can help to coordinate time schedules by presenting multiple options, where everyone can choose her/his preferred time slots.

**Description** To coordinate time and place of a meeting, a person creates a new survey, providing multiple time slots and one location and sets a predefined enddate, up until the survey is active. Then she/he sends the survey to all members of a community. Each member publishes its preferred options. If the enddate is reached, the reservation system matches the time slots. If there is a majority for one time slot, the room is booked automatically. For the room booking the organizer might need to deposit a specified amount of fungible objects (e.g., coins). If there are multiple options with the same amount of votes, it picks one of those slots randomly. In both cases all members are informed. If no one voted, the system informs the initiator, which can restart the process.

**Actors**

- Initiator - a member of the group that initiates the survey.
- Member - someone who is allowed to vote on the given options.

[1] `https://doodle.com`

**Preconditions**

- There is a contract for processing the answers and interacting with the reservation system and sending event-based notifications.
- The enddate must be in the future.
- The initiator can only select free timeslots for a room at a predefined date.
- Each member can only vote once.

**Postconditions**

- There is a room reservation in the system according to the result of the survey.
- The enddate lies in the past.
- Every member got a notification.
- In case of a failure (e.g., nobody voted for any of the given options), the initiator got a notification.

### 5.1.3 UC3: Q&A - Board

**Motivation** Discussion Boards represent a possibility to exchange interests, opinions and knowledge over the internet. Well-known representatives are Stack Overflow[2] or Quora[3]. They can be public or used in a private business environment. Technical discussion boards which realize a Q&A-style are helpful and be characterized by the fact that experts or experienced developers answer detailed technical questions.

**Description** A participant can start a new discussion thread by posting a question to a specified topic. After a new thread is opened, an expert can send her/his answer. Other experts can either join the discussion by giving another answer (that is in her/his opinion more correct) or by rating the other(s) answer(s) by providing an up or down vote. Questions can be filtered with keywords.

**Actors**

- Asker - A participant, interested in a topic (that may or may not be an expert).
- Expert - A participant with provable knowledge in the respective topic of the question, that can either give an answer or vote on other answers.

[2] https://stackoverflow.com
[3] https://www.quora.com

**Preconditions**

- There is at least one expert per topic.
- The expert has a knowledge certificate for the assigned topic.

**Postconditions**

- The question is answered by at least one expert.
- There are zero or more up or down votes on the answer(s).

### 5.1.4 Example Application (EA): A Virtual Media Library

**Motivation** As with all sharing communities, the intention behind sharing is cost efficiency and cooperation. A virtual shared media library benefits from a large amount of participants. For example, if everyone brings in three eBooks or movies (= media files) the amount of available media for everyone increases.

**General concerns** In a virtual shared media library different kinds of media like eBooks, eMagazines, music and movies are shared amongst participants (of a community). A participant is owner of a media file if she/he can prove ownership. The media files are available by URLs in a distributed file storage network. Each owner can distribute access keys to share media files.

**Restrictions** Media files are only visible and shared amongst users of a specified community (group). This means, that only a member of the community shall know about the existence or current state of a media file that is shared by a member of the same community. All members of this community need to agree on contribution or state changes (e.g., "available", "lent") of media files. Members of other communities shall not know about such details.

**EA-UC1: Borrow Media (e.g., eBook)**

**Actors**

- Owner $O$ - who can prove ownership and has access to the media file $m$
- Borrower $B$ - who wants to borrow the media file $m$ until a predefined due date $t$

**Description** A borrower $B$ requests a specific media file $m$ until a predefined due date $t$. If the owner $O$ wants to lend the file to borrower $B$, $B$ receives an exclusive valid access key from $O$. If the media file $m$ is (a) already borrowed by another person or (b) the owner wants to keep it, $B$ receives a notification.

**Preconditions**

- $O$ and $B$ are both members of the community.
- $O$ can prove ownership of $m$ and has the right to lend out $m$.
- The media file $m$ is stored on the distributed file storage.
- The state of $m$ is "available".

**Postconditions**

- $B$ holds an exclusive access key for $m$, that is valid until $t$.
- The state of $m$ is "lent".
- In case of (a) or (b) $B$ received a respective notification.

**EA-UC2: Preorder Media (e.g., eBook)**

**Actors**

- Owner $O$ - who can prove ownership of the media file $m$
- Borrower $B1$ - who wants to borrow $m$ until a predefined due date $t1$
- Borrower $B2$ - who has borrowed $m$ until a predefined due date $t2$

**Description** The media file is already borrowed by $B2$. Borrower $B1$ can place a preorder for the media file $m$ if the specified due date $t1$ is greater than $t2$. As soon as the media file $m$ is returned by $B2$ and its state is reset to "available", $B1$ gets a new access key to the media file and the state gets again set to "lent". Preorders are processed first-in-first-out (FIFO). If $O$ wants to keep $m$, $B1$ receives a notification.

**Preconditions**

- $O$, $B1$ and $B2$ are all members of the community.
- $O$ can prove ownership of $m$ and has the right to lend out $m$.
- The media file $m$ is stored on the distributed file storage.
- The state of $m$ is "lent".
- $B2$ holds an exclusive access key for $m$, that is valid until $t2$.
- $t1 > t2$

**Postconditions**

- $B1$ holds an exclusive access key for $m$, that is valid until $t1$.
- The state of $m$ is "lent".
- If $O$ wanted to keep $m$, $B1$ received a respective notification.

**EA-UC3: Return Media (e.g., eBook)**

**Actors**

- Owner $O$ - who can prove ownership of the media file $m$
- Borrower $B$ - who has borrowed the media file $m$ until a predefined due date $t$

**Description** The media file is already borrowed by $B$. Borrower $B$ can return the media file $m$ before $t$ if the due date $t$ is either today or in the future. By returning the book, the respective access key gets invalid.

**Preconditions**

- $O$ and $B$ are both members of the community.
- The state of $m$ is "lent".
- $B$ holds an exclusive access key for $m$, that is valid until $t$.
- $t \geq$ today

**Postconditions**

- $O$ has access to $m$.
- The state of $m$ is "available".
- The exclusive access key held by $B$ for the media file $m$ is invalid.

**EA-UC4: Renew Lending Period for Media (e.g., eBook)**

- Owner $O$ - who can prove ownership of the media file $m$
- Borrower $B$ - who has borrowed the media file $m$ until a predefined due date $t$

**Description** The media file $m$ is already borrowed by $B$. Borrower $B$ can renew the lending period for up to 14 days if $t \leq$ today. This is only possible if there are no pending preorders for $m$. The owner $O$ publishes a new access key with the updated validity.

**Preconditions**

- $O$ and $B$ are both members of the community.
- The state of $m$ is "lent".
- $B$ holds an exclusive access key for $m$, that is valid until $t$.
- $t \leq$ today
- There are no pending preorders for $m$.

**Postconditions**

- The state of $m$ is "lent".
- The exclusive access key held by $B$ for the media file $m$ is valid until $t + 14$ days.

**EA-UC5: Buy Media (e.g., eBook)**

- Seller - Owner $O$ who can prove ownership and has access to the media file $m$
- Buyer - Member $M$ who wants to buy the media file $m$

**Description** The media file $m$ is not borrowed by anyone. Owner $O$ publishes a price offer $p$ for $m$. Member $M$ pays the required amount of fungible objects (e.g., coins) to the address held by $O$. If $O$ receives the full amount, ownership of $m$ is transferred to $M$ automatically.

**Preconditions**

- $O$ and $M$ are both members of the community.
- $O$ and $M$ each have an account address and a wallet that holds a balance of fungible objects.
- $O$ can prove ownership of $m$ and has the right to lend out $m$.
- The media file $m$ is stored on the distributed file storage.
- The state of $m$ is "available".

**Postconditions**

- $M$ can prove ownership of $m$ and has the right to lend out $m$.
- The media file $m$ is stored on the distributed file storage.
- The state of $m$ is "available".

## 5.2 Community Blockchain Interaction Patterns

The community blockchain interaction patterns are derived from the community blockchain use cases outlined in Section 5.1. They can be divided into two types of patterns, namely Trading patterns and Technology patterns (see Figure 5.1).

The Trading patterns (P1-P3) are Transfer, Exchange and Sharing. Transfer describes an unidirectional transfer of fungible objects (e.g., coins). A bidirectional transfer of fungible objects and/or non-fungible objects is described in the Exchange pattern, where non-fungible objects represent either rights or obligations. Mostly ownership rights are traded in an exchange. For example in a purchase, fungible objects are transferred by the buyer in exchange to the ownership of non-fungible goods offered by a seller. A precondition of a purchase is an offer. The seller offers the non-fungible object (e.g., an eBook) to a specified price. The buyer transfers the respective amount of fungible objects (e.g., coins) to the sellers account. The purchase is realised if the seller's account receives the full amount of fungible objects and the non-fungible object becomes property of the buyer. The Exchange pattern can make use of the Trading pattern. Sharing concludes ownership of a non-fungible object by the lender. But in contrast to the Exchange pattern the ownership is never transferred. Instead access rights are provided in exchange of a fee or can be unidirectional provided by the lender, if for example the borrower can prove her/his membership in a community. Access rights are provided for a predefined time period. The Sharing pattern can make use of the Transfer pattern. It is slightly related to the Exchange pattern, but the transfer of ownership is strictly excluded.

The Technology patterns (P4-P6) are Chat, Election and Tracking. Not only are they used by community blockchain applications, they also describe basic mechanisms of the blockchain technology. The Chat pattern represents the messaging between peers (by sending and receiving transactions). The Election pattern provides a voting or agreement mechanism. This is similar to the consensus mechanism used in the blockchain system (i.e., to agree on next block proposals or next leader). The Tracking pattern uses cryptographic principles to efficiently store and retrieve data on the blockchain.

In the following sections, the six blockchain interaction patterns are described in detail:

### 5.2.1 P1: Transfer

A Transfer describes an unidirectional transfer of fungible objects (e.g., coins). Using cryptocurrency a defined amount of fungible objects (e.g., coins) is assigned to an account (or wallet). Each account is associated to and managed by a participant (similar to a bank account). Coins can be transferred from one (or more) account(s) to another account (simple transfer or joint payment). This is called a coin transaction. The participant can only transfer coins from her/his account. The access of an account is restricted (e.g., by password). Participants can be individuals or companies/groups (e.g., legal entities).

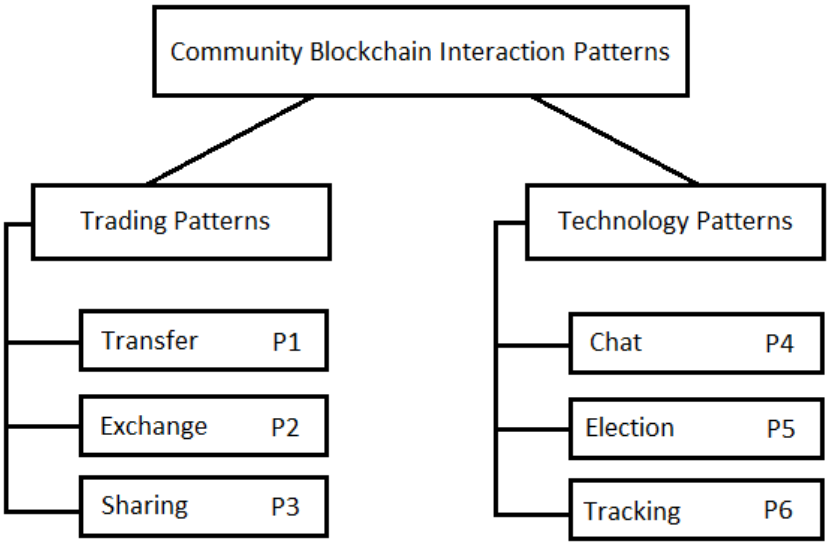


Figure 5.1: Overview of the proposed community blockchain interaction patterns, which can be divided into Trading patterns (P1-P3) and Technology patterns (P4-P6).

### 5.2.2 P2: Exchange

An exchange represents a bidirectional transfer of fungible and/or non-fungible objects (rights or obligations) between two participants. Participants can be individuals or companies/groups (e.g., legal entities). Exchanging the ownership of two non-fungible goods between two owners is called a swap. The non-fungible object can also represent debts. When granting a loan, a creditor transfers fungible objects in exchange for credit debt. Trading fungible objects in exchange to ownership of a non-fungible object is called a purchase (which is the most commonly used application of this pattern).

### 5.2.3 P3: Sharing

Sharing describes the shared usage of a non-fungible object between at least two participants. In contrast to an exchange, the owner keeps ownership of the non-fungible object. The access rights can be assigned to one (or more) participants for a pre-agreed period of time. For example access can be granted by an access key. This assignment is based on pre-defined rules (e.g., active release and withdraw of the object/usage rights).

### 5.2.4 P4: Chat

In a chat system each participant sends and receives messages (e.g., text messages). A chat has at least two participants, which all have the same view on incoming messages. This means that every participant receives all messages and the ordering of the incoming

messages is identical for all participants. Communication can be established between exactly two participants (1:1), between 1 sender and multiple receivers (1:n) or between multiple senders and receivers (m:n). The latter is comparable to a discussion forum. A chat is an append-only list of messages without a predefined end.

### 5.2.5 P5: Election

An election (or voting system) serves the democratic decision-making process by a defined number of participants. The right to vote is in-transferable and cannot be shared with others. One vote corresponds to one person. Therefore important factors are the identification of a person (actor), the persons right to vote, and the vote cast (asset) under preservation of election secrecy. It is not possible to link a participant to a vote. The result of an election is determined by counting of the votes cast. Participants can vote on a predefined set of options.

### 5.2.6 P6: Tracking

With tracking it is possible to store detailed information about non-fungible objects with a unique identifier. Either an external service (Oracle) certifies the correctness of the input data (e.g., recorded by a Sensor) or the data is entered manually by a trusted participant. Once data is recorded, it can not be changed. Recorded data can be queried by a participant by providing query properties.

## 5.3 Deriving Blockchain Patterns

This section shows how the community blockchain use cases described in Section 5.1 can be composed from the community blockchain interaction patterns described above (see Section 5.2). For this purpose, the use cases are broken down separately and the application of the six patterns is discussed in each context. It can be argued that the Technology patterns are applied in all use cases, since they are included in the basic mechanisms of a blockchain. This was deliberately not performed in the analysis, since only the applicability to the respective use cases should be shown here. The results of this analysis are outlined in Table 5.1.

**UC1: Acquiring and Sharing Resources** This use case is a combination of a collaborative purchase of a resource and its subsequent sharing. Therefore, the use case contains an exchange of fungible (e.g., coins) and non-fungible (e.g., ownership of the resource) objects, which is described in the Exchange pattern (P2). Since, the exchange includes a transfer of fungible objects by one or more participants, also the Transfer pattern (P1) is applied. The subsequent sharing of this resource can be realized by use of the Sharing pattern (P3). In this case, the community/group is the owner of the resource, and each individual member of the community/group is granted access on demand.

**UC2: Organizing Meetings with Automated Room Reservation** This use case consists of a voting on a timeslot and room. This voting can be described using the Election pattern (P5). The room booking can be realized using a Sharing pattern (P3). If the voting was successful, every participant gets a notification. This is represented by a 1:n communication in the Chat pattern (P4). The deposit of fungible objects (e.g., coins) includes the Transfer pattern (P1).

**UC3: Q&A - Board** In a Q&A - Board, questions are posted and answered by different participants. This can be realized using a n:m communication from the Chat pattern (P4). All participants see the questions and answers in the same order. Every expert can publish a new answer. Then other experts need to vote on the quality of the answer. For this voting with two options (up or down vote), the Election pattern (P5) can be used. To query and show all respective answers for a topic, the query mechanism described in the Tracking pattern is needed (P6).

**EA-UC1: Borrow Media (e.g., eBook)** In this use case, a media file is encrypted and stored on a distributed file storage. Each member of a community/group can borrow this file, by requesting an access key. This use case implements the Sharing pattern (P3). If a book is already borrowed by another member, the requesting member will be notified. This notification is realized by the Chat pattern (P4). Information about the media file (metadata) is stored in the blockchain, and can be retrieved by querying. Furthermore an Oracle needs to confirm, that the media file is stored in the distributed file storage. Both requirements are covered by the Tracking pattern (P6).

**EA-UC2: Preorder Media (e.g., eBook)** This use case consists of a preordering mechanism for media files. The preorder for a media file is represented by a message. Therefore it can be realized using the Chat pattern (P4). For the retrieval of the media file status ("available", "lent") the data must be query-able. This can be implemented using the Tracking pattern (P6). As soon as a media file is available, EA-UC1 is triggered (that uses the Sharing pattern (P3)).

**EA-UC3: Return Media (e.g., eBook)** In this use case, all participants will be informed by a notification. This can be implemented using a 1:n communication from the Chat pattern (P4). During the process of the state and access key update, query mechanisms will be used. This is part of the Tracking pattern (P6).

**EA-UC4: Renew Lendig Period for Media (e.g., eBook)** For renewing the lending period, the borrower triggers the sharing process again. Therefore the Sharing pattern (P3) is applicable. To retrieve details about the previous lending period and borrower, the querying process described in the Tracking pattern (P6) is used.

**EA-UC5: Buy Media (e.g., eBook)** In contrast to the other use cases from this example application, the ownership of a media file is transferred in this use case. To

| Use Case | Description | P1 | P2 | P3 | P4 | P5 | P6 |
|---|---|---|---|---|---|---|---|
| UC1 | Acquiring and Sharing Resources | ✓ | ✓ | ✓ | | | |
| UC2 | Meeting arrangement with room booking | ✓ | | ✓ | ✓ | ✓ | |
| UC3 | Q&A-Board | | | | ✓ | ✓ | ✓ |
| EA-UC1 | Borrow Media (e.g., eBook) | | | ✓ | ✓ | | ✓ |
| EA-UC2 | Preorder Media (e.g., eBook) | | | (✓) | ✓ | | ✓ |
| EA-UC3 | Return Media (e.g., eBook) | | | | ✓ | | ✓ |
| EA-UC4 | Renew Lending Period for Media | | | ✓ | | | ✓ |
| EA-UC5 | Buy Media (e.g., eBook) | ✓ | ✓ | | | | ✓ |

Table 5.1: This table shows the pattern composition for each use case. The brackets indicate, that there is a dependency between two use cases (i.e., Preorder Media and Borrow Media), and therefore the respective pattern does not represent a main component of this use case.

realize the purchase, the Exchange pattern (P2) is used. Due to the transfer of fungible objects from the buyer's wallet to the seller's wallet, the Transfer pattern (P1) is included. Since, only media files with state "available" can be purchased, respective data must be retrieved. This querying is included in the Tracking pattern (P6).

CHAPTER 6

# Pattern-based System Analysis

In the previous chapter, six community blockchain interaction patterns were derived from typical community blockchain use cases (e.g., file sharing, meeting arrangement). The analysis described in this chapter is conducted to show that these patterns can be implemented using existing permissioned blockchain systems. First, the methodology to classify permissioned blockchain systems according to the complexity of the solution for each pattern implementation is outlined in Section 6.1. Then, the components and code samples of the previously described permissioned blockchain systems (see Chapter 4 are analysed in Section 6.2 to show pattern applicability. A summary of the results of this analysis is provided in Chapter 7.

## 6.1 Methodology

A pattern-based analysis can help to identify significant differences between the systems. This becomes clear by classifying the systems in respect to the complexity of the implementation that is needed to address all requirements of a given pattern.

Mordinyi et al. [MKS10] differ between architecture limiter and breaker, which describe the architecture agility of a system dealing with new requirements. An architecture limiter shows the inability of the system to provide an efficient and simple solution when facing a new requirement. An architecture breaker indicates that in order to fulfil new requirements, the system demands for a complete re-evaluation of the architecture.

Since blockchain systems provide programming interfaces for flexible and complex solution design, the definition of an architecture limiter is not directly applicable in this context. Architecture limiters based on blockchain architectures can be seen as intended complex customized solutions (CS). Therefore, the criteria for architecture agility outlined by Mordinyi et al. is adapted in this thesis.

The differences between permissioned blockchain systems according to their solution complexity and the ability to cope with new requirements, are analysed using following three categories:

- Out of the Box (OOTB) - the system provides technical components or example code to simply and efficiently implement the pattern out of the box
- Customized Solution (CS) - the system provides a mechanism (e.g., smart contracts) to design customized solutions for the pattern (they must not need to be efficient and simple)
- Architecture Breaker (AB) - the system needs a complete re-design in order to provide the required functionality

## 6.2 System Analysis

In this section, the existing blockchain systems (described in Chapter 4) are analysed using the previously outlined patterns (see Section 5.2). This section shows how the patterns can be used for the technical assessment of blockchains. The analysis is based on a literature research on the latest official developer documentation. For each system, only one technical solution per pattern is outlined. It is possible that a pattern can be implemented otherwise.

### 6.2.1 Corda

The Corda Platform[1] was first released in October 2016 by R3. Corda is open source and published under the Apache 2.0 licence in a github repository[2]. It is written in Kotlin. The latest version at date is V 4.2. Corda provides extensive documentation[3], including code snippets and tutorials on how to start developing blockchain applications. The community contributing to Corda consists of over 200 companies, including major banks. In addition, Corda provides several developer tools, like DemoBench for local development and testing. The development is still ongoing.

**P1: Transfer** Corda's representation of a wallet is called a Vault[4]. It enables to transfer fungible assets (e.g., coins) from one Vault to another one. Such assets are technically represented by state objects. Each Vault maintains a list of unconsumed (=unspent) and consumed (=spent) states. To send a specified amount of fungible objects from one address to another address, unconsumed fungible state objects are combined (inputs) and a new output with the respective amount is created by a transaction. [Hea16]

[1] https://www.corda.net
[2] https://github.com/corda/corda
[3] https://docs.corda.net/head/index.html
[4] http://docs.corda.net/vault.html

Joint Payments, that involve multiple parties in the payment process, can be realized using the flow framework. A flow[5] enables complex multi-step processes by describing a private point-to-point communication in the Corda network. Such communication channels can be established between two or more parties, which must be specified by the initiator by including respective parties in the participant field of a state object. This flow concept can be used to collect signatures of different parties of a joint payment. [Hea16]

**P2: Exchange** A payment process between a buyer and a seller can be implemented using a simple flow: A seller will provide some offer by sending a reference to a state object (representing a non-fungible object) and an asking price. This offer will be sent to the buyer. The buyer will use this infos to create a transaction with two inputs and three outputs. It takes the referred state object and a fungible state object with the respective amount of coins as inputs. The transaction will create a new state, where the buyer gets to be the new owner. The other two outputs show the correct balances of both seller and buyer. This transaction is signed by the buyer and sent to the seller, which also signs it and returns it to the buyer again. Then the buyer sends it to the notary for validation. This TwoPartyTradingFlow is described in the offical docs[6] and is part of the Core implementation and can be used as template for further development of other exchange forms [7].

**P3: Sharing** Non-fungible objects can be represented by a LinearState in Corda[8]. Besides the contract reference and participants, the LinearState contains a unique identifier (linearId). This linear state identifier can include an URL to the encrypted resource. To assign a valid access key a participant will set an additional maturity date[9]. Like in a normal trade, the lender will make an offer providing reference details to this LinearState object (that represents the non-fungible object) and define additional options for the lending, like price and lending period. The borrower may accept the conditions by creating a signed transaction with the state object referencing the resource she/he wants to borrow, and the respective amount of coins as inputs. Then the state object containing an access key is given to B in a new state. This and the new balance states of lender and seller represent the transaction outputs. Corda supports event scheduling[10]. If the contract state (LinearState is a special form of a contract state) implements SchedulableState, the contract will be triggered by a specified event to automate a key invalidation procedure.

[5] http://docs.corda.net/key-concepts-flows.html
[6] https://docs.corda.net/flow-state-machines.html
[7] https://github.com/corda/corda/tree/master/finance/workflows/src/main/kotlin/net/corda/finance/flows
[8] https://docs.corda.net/financial-model.html
[9] https://docs.corda.net/api-states.html#user-defined-fields
[10] https://docs.corda.net/event-scheduling.html

**P4: Chat** In Corda, there is no general message or transaction broadcast, like in typical blockchains. Transactions are shared between specified parties in a point-to-point communication (using flows). A sender needs to define a list of participants for each state object, which are either involved in the verification process or need to sign the transaction in order to correctly interact in the flow. Still, there is a possibility to create a broadcast message. The BroadcastTransactionFlow can be used to reach an additional specified list of receipients, which act as observers[11] only. This concept of observer nodes can be used for 1:n and m:n communication.

**P5: Election** A main goal of Corda is to provide legal binding in a network of untrusted participants. To ensure this property, an entity is linked to exactly one public key. This is done by identity providers. The privacy model is based on flows (point-to-point communication) between participants. It involves notary pools to ensure validity of a transaction. This includes that certain information is shared with validating peers. Hence, unlink-ability is not guaranteed by design. At least one peer will know who voted for what, unless additional privacy protocols are used.

**P6: Tracking** Corda uses services to approve correctness of certain facts that are included in a transaction. Such services are called Oracles[12]. Privacy dependent properties can be hidden from these services using Transaction tear-offs[13]. Once a state object is recorded in the Vault, the encapsulated data or information can be queried. For this purpose, Corda provides a Vault Query API and other approaches, like custom JPA queries. One can for example specify filters by definition of a QueryCriteria object[14].

### 6.2.2 Hyperledger Fabric

Hyplerledger Fabric[15] is one of the frameworks contributed under the Hyperledger project. The first alpha release was published in March 2017 by IBM. Hyperledger Fabric is open source under the Apache 2.0 license. The code is available via git repository[16] and written in Golang. It is an active project (meaning that there is still ongoing development) with an extensive documentation[17], including tutorials and code samples. The latest stable release is version V1.4.1, but there is also a newer alpha version V2.0 available since April 2019. The Hyperledger Fabric Chaintool[18] supports chaincode (i.e., smart contract) developers.

[11] https://github.com/roger3cev/observable-states
[12] https://docs.corda.net/key-concepts-oracles.html
[13] https://docs.corda.net/key-concepts-tearoffs.html
[14] https://docs.corda.net/api-vault-query.html
[15] https://www.hyperledger.org/projects/fabric
[16] https://github.com/hyperledger/fabric/tree/master
[17] https://hyperledger-fabric.readthedocs.io/en/latest
[18] https://github.com/hyperledger/fabric-chaintool

**P1: Transfer** Wallets in Hyperledger Fabric store identities, but do not hold any other state, like a balance of coins. Identities are represented by X.509 certificates. A certificate is issued by a trusted root authority and is used to assign access privileges for different application frameworks.

Any fungible object must be stored in the state of a chaincode. The chaincode defines functions to issue and transfer assets from one address to another. Assets are represented by key-value pairs. It is possible to create fungible objects with a Bitcoin-like UTXO model. In fact, the newest alpha release 2.0 provides the FabToken[19], which represents fungible tokens in a UTXO model. Respective functions, like an asset transfer, can be triggered by sending transactions.

A contract code defines rules of payment strategies. Respective transactions for joint payments need to be signed by multiple parties. To correctly transfer coins from one address to another one, the transaction is validated by endorsing peers. Endorsing peers are specified by an endorsing policy. In case of a joint payment, it make sense to assign all involved parties in the endorsing policy.

**P2: Exchange** A purchase can be described by a smart contract (chaincode). A possible endorsement policy specifies buyer and seller to verify the transactions for the purchase. The respective data model would provide separate application wallets for buyer and seller, as well as contracts for coin issuing and transfer. A new non-fungible object needs to be created in another contract. The contract code for the purchase needs a function to initiate an offer by a seller. Furthermore functions to query the offers and to buy an asset can be called by a buyer. In this way, other exchange forms can be created.

**P3: Sharing** Sharing rules and logic can be described in a smart contract. To borrow a non-fungible asset, contracts for issuing and transfer of fungible and non-fungible objects need to be deployed. The endorsing policy for sharing includes borrower and lender.

**P4: Chat** Hyperledger Fabric uses channels for private communication. Each channel is associated with a separate blockchain ledger (holds world state and transaction log). For each channel, new chaincode needs to be deployed together with endorsement policies and MSPs. Therefore another concept of private data collections[20] was introduced by V1.2 of Hyperledger Fabric. Such collections allow subsets of participants in a channel to commit, query and endorse private data. To store private data independently to public data, each peer maintains a private state database (also called SideDB). Only authorized participants receive private data. This applies also to ordering peers. The private data is hashed. This hash is stored in the channel's blockchain ledger. In contrast to public data, the private data is shared by point-to-point communication (not in blocks). This

[19] https://hyperledger-fabric.readthedocs.io/en/latest/whatsnew.html#fabtoken

[20] https://hyperledger-fabric.readthedocs.io/en/release-1.4/private-data/private-data.html

feature can be used, when a private 1:1 or 1:n chat communication is intended. For a discussion board application with m:n communication a channel is sufficient. Private data collections raise new problems, as the data sent, is not included in blocks.

**P5: Election** In Hyperledger Fabric each wallet can represent a set of different identities. They need a certificate, which is signed by a trusted root authority. They store an associated public/private key pair. A new identity can be issued for a voting process. The voting is then processed in a channel. Voting logic is defined in a contract that ensures that each participant votes only once. When a voting process finishes, the result can be transmitted to another channel for further processing. The voting identities can be deleted after that.

**P6: Tracking** Assets are represented in the world state. To query or filter those assets, the query logic can be described in smart contracts. Smart contracts designed to manipulate the assets, usually also contain query logic for these assets.

### 6.2.3 Hyperledger Sawtooth

Hyperledger Sawtooth[21] is another framework contributed under the Hyperledger project. It was first released in January 2018 by Intel. The source code is available via github repository[22] and written in Python 3. The framework is open source under the Apache 2.0 license. The latest release at date is V1.1.5. Hyperledger Sawtooth provides several code examples of transaction families as well as an extensive documentation[23]. It is an active project, with ongoing development.

**P1: Transfer** In Hyperledger Sawtooth, transaction families describe all possible transactions that can occur in an application. It represents a data model and rules. This rules describe how to interpret the transaction payload. An example of a transaction family is Smallbank[24]. It describes a simple account datatype, that is similar to a Bitcoin wallet. This account consists of a customer_id and a balance (amount of coins owned by this customer). The Smallbank transaction family furthermore specifies different transactions for account creation and balance modification. One of them can be used to transfer coins from a sender's account to a receiver's account. An instance of this transaction triggers the respective function in the transaction processor, that will actually perform the coin transfer.

Joint payments can be implemented by encapsulation of multiple coin transfer transactions into a batch. Each coin transfer transaction describes a simple transfer of coins to the receivers account. All transactions are collected and then packed into a batch. If one of

[21] https://sawtooth.hyperledger.org

[22] https://github.com/hyperledger/sawtooth-core

[23] https://sawtooth.hyperledger.org/docs/core/releases/latest

[24] https://sawtooth.hyperledger.org/docs/core/releases/latest/transaction_family_specifications/smallbank_transaction_family.html

the transactions is invalid the batch will fail, otherwise all transactions are executed at once.

**P2: Exchange** Hyplerledger Sawtooth provides three explicit examples that illustrate blockchain applications in varying contexts. One of them is the Marketplace[25]. It describes a more complex data model than Smallbank. Besides simple coin transfer, it supports purchasing and other asset exchange forms. Each asset is described by an asset type. Ownership is represented by holdings. A non-fungible object could for example be specified by a holding with an instance-count set to 1 of a consumable, non-devisible and restricted asset. The Marketplace provides ExchangeOffers and SellOffers that can be used to perform a purchase.

**P3: Sharing** The examples from the Exchange pattern (P2) could be used to implement sharing logic. Since there is no exchange of ownership on the shared resource (any non-fungible object), the owner of the resource creates an access key and transfers the ownership of this key to the borrower. Hyperledger Sawtooth does not provide time-triggered events. Therefore the application has to issue events that can be tracked by subscription[26].

**P4: Chat** Transaction payload in Hyperledger Sawtooth is represented by a byte array. It can be used to store messages (e.g., text message) for chat communication. By default, transaction payload can be accessed by all permissioned nodes in the validator network[27]. However, there is an alpha version[28] available that introduces private transaction families. They can be used to ensure privacy on the transaction data by privacy-based access control. It uses a hardened version of the transaction processor, that utilizes the Trusted Execution Environment (TEE) from Intel SGX. This enables decryption and validation of transactions in an isolated environment, so that the memory cannot be accessed by any other process[29].

**P5: Election** The blockchain project Bitagora[30] implements a real-world blockchain-based voting platform built on Hyperledger Sawtooth. It describes a voting protocol, that supports unlinkability. A pollster defines the poll and registration requirements, which are checked by the certifier against the provided information from a voter. If a voter is allowed to participate, the certifier creates a key pair and sends the respective

[25] https://sawtooth.hyperledger.org/examples/marketplace.html

[26] https://sawtooth.hyperledger.org/docs/core/nightly/master/app_developers_guide/event_subscriptions.html

[27] https://sawtooth.hyperledger.org/docs/core/releases/1.0/architecture/permissioning_requirement.html

[28] https://github.com/hyperledger-labs/private-transaction-families

[29] https://github.com/hyperledger-labs/private-transaction-families/blob/master/SPECIFICATION.md

[30] https://www.hyperledger.org/blog/2018/09/20/bitagora-a-decentralized-voting-platform-built-on-hyperledger-sawtooth

private key and a signed token to the voter. This token is generated by the certifier using the public key of the created key pair. The voter can use this key to cast a ballot.

**P6: Tracking** Hyperledger Sawtooth provides an example of a supply chain transaction family [31]. The corresponding supply chain transaction processor supports the storage of records, to keep track of the ownership of an asset. The example also provides an SupplyChain REST API that can be used to retrieve specified records. The example syncs blockchain data to a local database, to provide better query options.

### 6.2.4 Multichain

Multichain[32] was first released in August 2017 by Coin Science Ltd. It is published as open source project under the GNU General Public Licence version 3 (GPLv3). For further support and redistribution, they provide a pricing model that includes a commercial license and Service Level Agreements (SLAs). Multichain's source code is written in C++ and published in a github repository[33]. The latest release version is V2.0.1. Multichain provides a documentation[34], including tutorials and code snippets, and a well-supported developer Q&A board. There are different developer tools available, like the Multichain Explorer, that is used to track blockchain activities. YobiChain[35] enables a quick development of Multichain-based blockchain applications.

**P1: Transfer** Since Multichain is based on Bitcoin Core, it supports a native currency. By default this native currency is not used. This is configurable by setting other blockchain parameters[36]. There is another mechanism in Multichain to issue fungible assets[37] using the 'issue' command. There are certain addresses with permission for issuing new assets that usually have admin rights. Multichain distinguishes between open and closed assets. Closed assets can only be issued once. Open assets can be reissued if more of the same unit is needed. Assets created by an issue function can be transferred in transactions, using the UTXO model known from Bitcoin. A sender specifies which assets she/he wants to transfer. Assets in this context can be native currency or issued assets or a mix of both. The sender specifies the receiver address and the respective amount of assets. Each wallet in Multichain can have multiple addresses. Permissioning is based on a handshake (see Section 4.4).

To realize joint payments in Multichain, one party needs to create and sign a raw-transaction including the UTXO. This transaction will be sent to one other party, which

[31] https://sawtooth.hyperledger.org/docs/core/nightly/0-8/examples/supplychain/supplychain_transaction_family.html
[32] https://www.multichain.com
[33] https://github.com/MultiChain/multichain
[34] https://www.multichain.com/developers
[35] https://github.com/Primechain/yobichain
[36] https://www.multichain.com/developers/blockchain-parameters
[37] https://www.multichain.com/developers/asset-reissuance

adds its UTXO and signs it, and so on. Appending is enabled by setting sighashtype to SIGHASH_ALL |SIGHASH_ANYONECANPAY[38,39].

**P2: Exchange** Non-fungible objects can be represented by issuing a closed fungible asset with a single unit. Each asset has a unique asset reference. A purchase can then be conducted using atomic exchange transactions[40]. A seller creates an offer by issuing a new exchange transaction calling createrawexchange() function[41]. This offer holds a reference to the non-fungible asset and the expected price. The buyer calls appendrawexchange(), that issues a transaction. This transaction contains the output of the seller's function call and an UXTO with the respective amount of fungible assets to buy the non-fungible resource. The output of this function call is broadcast to the network, resulting in the completed trade.

**P3: Sharing** The sharing of a non-fungible object can be implemented using encrypted keys that provide access to the shared resource. The resource and an access key can be represented by a non-fungible objects. The access key is provided by a simple transfer. Since, Multichain does not support time-triggered events, according logic for the invalidation of the access key needs to be established on the application layer.

**P4: Chat** Communication in Multichain is enabled by a concept called Streams[42]. The content of a stream can either be stored publicly on the blockchain ledger or in a private off-chain storage. In the second case, only a referencing hash of the data is included in a transaction. Each stream represents an append-only list of items, where each item corresponds to a blockchain transaction. It can be published by one or more participants. All participants need to sign this item. For querying purposes, a key must be stored for each item. Furthermore, each item encapsulates some data (e.g., text message) and a timestamp. There are two kinds of streams: open streams and closed streams. While the first ones can be written by anyone with permission to create transactions, the latter are maintained by administrators, who grant access to a specified group of participants. Permitted nodes need to subscribe to a stream in order to get access. Both, public and private chats are possible using streams.

**P5: Election** Stream filters[43] can be used for validation in Multichain. For an election they might determine if someone is allowed to participate in the voting process. A private stream can be used for a voting. Since everyone with permissions for that stream will be able to link addresses to proposals, a new address (that is only used in this particular

[38] https://www.multichain.com/qa/11531/is-it-possible-to-make-a-joint-transaction
[39] https://bitcoin.org/en/developer-guide#signature-hash-types
[40] https://www.multichain.com/developers/atomic-exchange-transactions
[41] https://www.multichain.com/developers/json-rpc-api
[42] https://www.multichain.com/blog/2016/09/introducing-multichain-streams
[43] https://www.multichain.com/developers/smart-filters

voting process) should be created by each voter. This is possible, since a wallet can have multiple addresses. An administrator needs to assign the voters address to the voting stream.

**P6: Tracking** A permissioned oracle service can store an item in a stream. The item data holds a reference and detailed information about a non-fungible object. As each item must be signed by its publisher, the oracle will confirm the correctness of the data. Tracking is ensured by the querying options in Multichain. They can filter items according to publisher and key. It is also possible to order items, and filter the number of items issued by a publisher and the number of items with the same key.

### 6.2.5 Quorum

Quorum[44] was first released in November 2016 by J.P. Morgan Chase. The project represents an open source fork of the go-ethereum[45] implementation. While the go-ethereum binaries are licensed under the GPLv3, the go-ethereum library is published under the GNU Lesser General Public License Version 3 (LGPLv3). The source code is written in Golang and provided in a public github repository[46]. The project provides a wiki[47] and two examples that help developers get started quickly. The latest version V2.2.3 was released in April 2019. There are several development tools, such as Cakeshop (an integrated development environment and SDK with graphical interface) a Blockchain Explorer (to view private transactions) and the QuorumNetworkManager (for network creation and management). Quorum is under active development.

**P1: Transfer** Since Quorum is based on Ethereum, it supports transfer of fungible objects (e.g., coins, tokens) by design. Quorum's Ether can be used similar to how it is used in Ethereum, but it is not related to the cryptocurrency Ether on the Ethereum main network, and therefore can not be used interchangeably. Ethereum's Ether is an incentive for mining in the PoW consensus, but does not have the same value in private networks. But the relation to the general design of Ethereum enables to create contracts that implement the ERC-20 token standard in Quorum. It describes an interface for smart contracts to implement fungible tokens. The standard includes functions for token transfer with a specified amount to a given receiver address. It is also possible to determine the total balance of accounts. Other functions allow withdrawing from given accounts (with owner permissions). Properties can define the maximum supply of tokens.

Quorum uses network permissions to restrict network access. This is controlled by the network manager component. Permissioned nodes have a wallet. Each wallet can have multiple associated addresses (part of a public key).

[44] https://www.goquorum.com
[45] https://geth.ethereum.org
[46] https://github.com/jpmorganchase/quorum
[47] https://github.com/jpmorganchase/quorum/wiki

To enable joint payments in Quorum, a smart contract needs to be defined. Like Ethereum smart contracts, any contract in Quorum has its own address. This can be used to collect fungible tokens from different parties. The creator of a contract must specify the required amount, a receiver address and an optional maturity date. If the required amount is reached, the contract can automatically send the full amount to the specified receiver address. If the required amount is not transferred by the end of the maturity date, the collected tokens are returned to the initial senders. Smart contracts can emit events on execution[48], but a time scheduled event is not implemented by design. This needs to be triggered by a service component (e.g., when the maturity date is reached).

**P2: Exchange** Non-fungible tokens are described in another Ethereum standard, namely the ERC-721[49]. It specifies ownership of the token, and provides functions to transfer this ownership. Each token is identified in the blockchain ledger by a unique tokenId. The most well-known application of ERC-721 tokens is Cryptokitties[50].

Rules for a purchase can be defined in a smart contract. ERC-721 tokens represent the traded resource (non-fungible object) that is owned by a seller. The seller needs to deploy a contract (similar to an offer), where she/he defines the expected price for the resource(s). If the buyer sends the respective amount, the contract will initiate the ownership transfer from the seller to the buyer. Since buyer and seller can inspect the contract logic at any time, and once a contract is published on the blockchain, it cannot be changed by either of the parties, both can be sure, that this logic does not contain any mistakes (establishes trust). Similar contracts can be used for other exchange forms.

**P3: Sharing** The rules for sharing are defined in a smart contract. Sharing includes both token standards (ERC-20, ERC-721). An ERC-721 token is transferred from the lender to the borrower. A lender might deploys a contract including the token for a shared resource. This contract defines legal rules for usage (e.g., the borrower is not allowed to dispense the token) and a routine to return ownership after a predefined time period. The function to return ownership can include a respective condition, but must be called by a transaction.

**P4: Chat** Public transactions in Quorum are structured like standard Ethereum transactions. Transaction size is a configurable parameter in Quorum's genesis block[51]. By default it is limited to 64kb, but it can be resized by a blockchain operator to a maximum of 128kb. Transactions contain a data field, where chat messages can be encapsulated. For 1:1 messaging, private transactions will be sufficient to enable privacy in Quorum. The payload of a private transaction is encrypted by the enclave and therefore

[48] https://solidity.readthedocs.io/en/latest/structure-of-a-contract.html#events

[49] http://erc721.org

[50] https://www.cryptokitties.co

[51] https://github.com/jpmorganchase/quorum/wiki/Using-Quorum#configurable-transaction-size

only visible to participants, whose public keys are stored in the privateFor parameter of the transaction. Other participants will only see a hash of this data in the respective field of the transaction.

**P5: Election** Similar to Hyperledger Fabric, a smart contract can be used to implement voting logic in Quorum. As already mentioned before, Quorum provides private transactions. This feature can be used to specify the participants, allowed to vote in the voting process. There is a good example of a voting contract given in the official Solidity documentation[52]. The problem in this implementation is the remaining linkability of participants and their proposals. This problem is addressed in other solutions by the use of tokens. Issuing fungible tokens result in another problem, namely possible double spending. Issuing non-fungible tokens will result in the same linkability problem, even if each participant creates a new address that is used only in this voting process. Zero-knowledge proofs might be used to address this problem. Quorum introduces a Proof of Concept for a protocol called ZSL (zero-knowledge security level), which uses z-tokens[53]. This enables transactions without revealing information about the sender.

**P6: Tracking** A permissioned oracle service can interact with smart contracts using transactions. Events can be used to log new data included by oracles. Detailed information about contracts and transactions can be filtered using Ethereum-based JSON-RPC[54]. This can be tedious, as contract states are stored in bytecode.

### 6.2.6 Tendermint

Tendermint [55] was first published in August 2016 by All in Bits, Inc. Since then, the framework is constantly under development and still tagged as alpha software. The latest version V0.30.4 was released in April 2019. Tendermint is open source and published under the Apache 2.0 license. The source code is written in Golang and publicly available in a github repository[56]. There exists an extensive developers documentation[57], including code snippets and examples. Tendermint provides tools for benchmarking, monitoring and signers verification. Example applications and blockchain solutions, like the Cosmos SDK or BigchainDB, are carried under the Tendermint ecosystem[58].

In contrast to other presented systems in this chapter, Tendermint provides a flexible application platform, that consists of a consensus engine and a respective communication interface for applications (ABCI). As transaction design and validation is outsourced to the application developer, Tendermint provides a flexible model for permissioned

[52] https://solidity.readthedocs.io/en/latest/solidity-by-example.html#voting
[53] https://github.com/jpmorganchase/quorum/wiki/ZSL
[54] https://github.com/ethereum/wiki/wiki/JSON-RPC
[55] https://www.tendermint.com
[56] https://github.com/tendermint/tendermint
[57] https://tendermint.com/docs
[58] https://tendermint.com/ecosystem

blockchain systems on top of the consensus engine. This flexibility comes with the trade-off, that application developers have to re-implement basic functionalities that other permissioned blockchain systems integrate in their design. Therefore, the Tendermint developer team introduced the Cosmos-SDK[59]. It allows customers fast development of Tendermint-based blockchain applications. As a consequence, the evaluation of the pattern applicability to the Tendermint system is extended to different Tendermint Applications from the Tendermint Ecosystem[60].

**P1: Transfer** In previous versions, Tendermint had a native cryptocurrency. It was used by peers to participate in a Proof of Stake based consensus. Newer versions implemented a BFT-based protocol, that does not need any form of coin issuance as incentive for block creation. Tendermint provides a platform for the implementation of distributed applications. It is a flexible approach that enables applications to implement their own cryptocurrency. An example application is Ethermint[61], which implements the Ethereum Virtual Machine (EVM), that runs on a Tendermint infrastructure.

A tendermint application needs to implement a cryptocurrency or fungible object trading platform from scratch. An example application is Basecoin[62], which is a simple cryptocurrency based on Bitcoin's UTXO model. It provides functions to initially create accounts and different kinds of coins (fungible objects). A simple coin transfer is implemented in a SendTx function. Transactions contain multiple inputs and outputs. Validation logic is called when the peer receives CheckTx or DeliverTx messages. The payment is executed only when a DeliverTx is received. A bank module[63] in the Cosmos-SDK provides similar functionality.

The logic for joint payments is not implemented in the above mentioned Basecoin application. In order to enable joint payments, both validation logic and the structure of transactions need to be adjusted.

**P2: Exchange** The Cosmos-SDK-Tutorial shows how a purchase can be realized on the Tendermint blockchain. The application represents a basic nameservice[64]. A name is a non-fungible object with an owner and price, that can be traded. The Cosmos-SDK bank module provides a simple cryptocurrency to enable trading. Unused names can be bought. A buyer sends a respective message that specifies the name (non-fungible object), a bid (amount in coins) and her/his account address. A function automatically executes the trade after checking validity. Other exchange forms can be implemented similarly.

[59] https://github.com/cosmos/cosmos-sdk
[60] https://tendermint.com/ecosystem
[61] https://github.com/cosmos/ethermint
[62] https://github.com/tendermint/basecoin
[63] https://github.com/cosmos/cosmos-sdk/tree/develop/x/bank
[64] https://github.com/cosmos/sdk-application-tutorial/tree/master/x/nameservice

**P3: Sharing** Similar to the above described nameservice application, a platform for sharing of non-fungible objects can be implemented using Tendermint. The respective rules for lending and automated transfer of ownership after a predefined lending period can be described in the application.

**P4: Chat** As Tendermint does not specify a structure for transactions, any application can send text messages using transactions. TMChat[65] is a Java application, that shows how a simple chat can be implemented with Tendermint. Messages are broadcast between participants. For private 1:1 communication, message encryption needs to be integrated.

**P5: Election** Tendermint can be used to implement various other consensus protocols on top of the Tendermint blockchain. This is shown by the Ethermint[66] example application, which implements EVM on Tendermint. Therefore, it is possible to write an application to conduct a voting. Message encryption can be used for privacy issues. Still, unlinkability can not be guaranteed.

**P6: Tracking** Tendermint provides the ABCI command 'query'[67] to filter informations about the application state from the local blockchain storage of a peer. Therefore, query messages are executed on a local peer and are not broadcast in the Tendermint network. Each query command can specify a path, that needs to be defined when an application creates transaction data. For this path, application specific types are recommended.

[65] https://github.com/wolfposd/TMChat
[66] https://github.com/cosmos/ethermint
[67] https://tendermint.readthedocs.io/en/v0.21.0/abci-spec.html#query

CHAPTER 7

# Results & Discussion

## 7.1 Pattern-based System Analysis Results

The results of the pattern-based system analysis conducted in Chapter 6 are summarized in Table 7.1. According to the complexity of the solution one can see, that the Transfer pattern (P1) and the Exchange pattern (P2) can be built out of the box (OOTB) in most of the systems. For many systems example code to implement a purchase is outlined in a tutorial section of the specification documents. The Exchange pattern (P2) includes other trading forms, that need an adaption of the code examples. Therefore in Table 7.1 the respective systems are marked with OOTB/CS. A majority of the systems do not support time-triggered events in the Sharing pattern (P3). Time-triggered events are needed for an automatic return of a shared asset or the automatic invalidation of an access key. Therefore, this pattern is solved mainly by customized solutions (CS) on the application layer. With the exception of this requirement, the Trading patterns (P1-P3) are well supported throughout the different systems. This might be reasoned by the fact, that permissioned blockchains evolved from cryptocurrencies, such as Bitcoin or Ethereum. Hyperledger Fabric and Tendermint need more customization for the implementation of the Trading patterns. This may be, because they offer more flexibility in the custom design of transactions than other blockchain solutions. Tendermint for example is designed to offload the business logic to the application layer. Hyperledger Sawtooth provides even more logic and flexibility on validation of transactions (due to their concept of transaction families), but provides code examples which realize the Transfer pattern (P1) and the Exchange pattern (P2).

Looking at the Technology patterns (P4-P6), it is noticeable that the Chat pattern (P4) is often supported out of the box. In most systems, there is a mechanism that enables the private exchange of messages. However, a major problem is the implementation of the Election pattern (P5). Unlinkability can only be set up with custom solutions. This can lead to new problems. It almost always needs an administrator. This contradicts the

| System | P1 | P2 | P3 | P4 | P5 | P6 |
|---|---|---|---|---|---|---|
| Corda | OOTB | OOTB/CS | OOTB | CS | CS | OOTB |
| Hyperledger Fabric | OOTB | CS | CS | OOTB | CS | CS |
| Hyperledger Sawtooth | OOTB | OOTB/CS | CS | OOTB | CS | CS |
| Multichain | OOTB | OOTB/CS | CS | OOTB | CS | CS |
| Quorum | OOTB | OOTB/CS | CS | OOTB | CS | CS |
| Tendermint | OOTB | CS | CS | OOTB | CS | CS |

Table 7.1: This table shows the classification of the analysed blockchain systems. For each pattern the solution can be provided out of the box (OOTB). Otherwise it might need a costlier customized solution (CS). The classification methodology is outlined in Section 6.1.

democratic and decentralized principles of a blockchain. The advantage of a blockchain over conventional systems is questionable in this context. However, there is ongoing research on privacy protocols and zero-knowledge proofs that can be established on top of the blockchain systems. Quorum offers at least one sample for the use of a zero-knowledge proof based protocol. However it is used in a trading context. There is also an example application in Hyperledger Sawtooth, that describes how to implement unlikability in real-world Elections. For both, Quorum and Hyperledger Sawtooth a customizable solution for the Election pattern (P5) is available. They provide an example of a privacy protocol that can be customized and implemented on top of the underlying blockchain.

The Tracking pattern (P6) is very well supported by Corda. They offer components to integrate oracles and provide a sophisticated query language. Other systems often do not provide an efficient query support. Instead they provide a basic query functionality, that can often be customized (e.g., in Quorum queries need to be defined in contracts). Hyperledger Sawtooth provides a Supply Chain example, but to allow efficient querying, the blockchain state is copied into a local database. Especially the querying of off-chain storage is a problem in permissioned blockchains, and must be considered in the application design. Useful metadata must be stored on-chain for efficient data retrieval.

## 7.2 Universal Applicability of Interaction Patterns

In Chapter 4, the applicability of the proposed blockchain interaction patterns on community blockchain use cases is discussed. They only cover a small spectrum of possible applications. A collection of current blockchain applications across different industries is for example given by [CDP19] and [ARF+19] (see Section 3.2). Figure 7.1 shows the classification of blockchain applications described by [CDP19].


Figure 7.1: Application areas for blockchain-based applications outlined by Casino et al. [CDP19]. This figure is a reprint from [CDP19].

In this section, the proposed community blockchain interaction patterns are discussed in respect to different application areas, namely Supply Chain Management (SCM), Cryptocurrencies, Crowdfunding, Energy Trading, Educational Certificates, Electronic Health Records (EHR) and Carsharing (see Section 3.2 for more use cases). The results are summarized in Table 7.2.

**Supply Chain Management (SCM)** SCM is an important application throughout different industries. It describes the planning and management process used in logistics. In this application, the blockchain provides more accountability and transparency than conventional systems. It is used to identify and observe a product or raw materials along

the entire life cycle, i.e., from the source to the manufacturing process through to delivery to the customer. This use case implements the Tracking pattern (P6). The data can be stored on a blockchain ledger using oracles (e.g., a certified smart sensor that captures the current temperature for each product at different times).

**Cryptocurrency/Crowdfunding** The most popular use case of a blockchain is a cryptocurrency, like Bitcoin or Ethereum. Another blockchain application in the financial sector is crowdfunding, as outlined by [Swa15]. The concept is known for decades, but crowdfunding platforms like kickstarter[1] and rocketHub[2] created a new trend. Crowdfunding allows individuals or small companies to quickly realize a business idea by raising money. Instead of searching for a major investor, many small investors can contribute to provide the full amount needed. They support the business idea by pre-financing it. Both cryptocurrencies and crowdfunding can be realized using the Trading pattern (P1). It includes an unidirectional transfer of fungible objects (e.g., coins) and a joint payment. If the investors get a product in return to a successful funding project, the more complex Exchange pattern (P2) can be used.

**Energy Trading** In [ARF+19] the authors describe a variety of use cases in the energy sector. One application in this context is mentioned by [CDP19]. It is the Emission Trading and Reputation platform proposed by Khaqqi et al. [KSHK18]. The goal of the application is to reward companies, that accomplish emission reduction, by giving them a high reputation and preferring their offers in a permit trading platform. The system uses certified smart meters to monitor $CO_2$ emissions produced by the companies. The trading platform can be implemented using the Exchange pattern (P2). Since permits can be bought by other companies that produce too much $CO_2$, the Transfer pattern (P1) is applicable. The use of certified smart meters to track the $CO_2$ emissions, indicates the need for the Tracking pattern (P6).

**E-Voting** Another area where blockchain applications can provide trust is the e-governmental sector. Especially for the transparent conduction of elections, the blockchain is a possible solution. The Election pattern (P5) is designed in respect to the requirements of a real-world election. Therefore it can be applied in the context of E-Voting.

**Educational Certificates** An efficient verification system of educational certificates is a potential blockchain application. It can serve employers or educational institutions in a job application process. This use case can be realized using the Tracking pattern (P6).

**Carsharing** Another popular use case is Carsharing. It can be realized using the Sharing pattern (P3). Since the carsharing companies charge per minute or hour, the Transfer pattern (P1) is applicable. Modern carsharing apps provide enhanced functions,

[1] `https://www.kickstarter.com`
[2] `http://www.rockethub.com`

| Application | P1 | P2 | P3 | P4 | P5 | P6 |
|---|---|---|---|---|---|---|
| Supply Chain Management (SCM) | | | | | | ✓ |
| Cryptocurrency/Crowdfunding | ✓ | ✓ | | | | |
| Energy Trading | ✓ | ✓ | | | | ✓ |
| E-Voting | | | | | ✓ | |
| Educational Certificates | | | | | | ✓ |
| Carsharing | ✓ | | ✓ | | | ✓ |
| Electronic Health Records (EHR) | | | | ✓ | | ✓ |

Table 7.2: This table shows the applicability of patterns in the selected blockchain applications.

like displaying nearby cars on a map. They keep track of the car via GPS. Therefore, the Tracking pattern (P6) can be used for carsharing applications.

**Electronic Health Records (EHR)** There are several blockchain-based applications in the medical sector. One of it is MedRec [AEVL16]. It provides access transparency of the sensible patient data. EHR can be realized using the Tracking pattern (P6). Another approach is a platform for personal treatment. For example by providing online medical advise by a doctor. This can be implemented by a private 1:1 communication described in the Chat pattern (P4).

CHAPTER 8

# Conclusion

## 8.1 Summary

In this thesis, six community blockchain interaction patterns were derived from basic community blockchain use cases. The described interaction patterns are Transfer (P1), Exchange (P2), Sharing (P3), Chat (P4), Election (P5), and Tracking (P6). While the first three represent Trading patterns, the other three describe Technology patterns. Trading patterns (P1-P3) describe an uni- and bidirectional exchange of fungible and non-fungible assets, and the time-limited transfer of shared usage rights. The Technology patterns (P4-P6) are on one hand application-based patterns for collaboration on shared data. On the other hand they describe core patterns of the underlying blockchain technology (exchange of messages, agreement amongst a group of participants, and efficient data storage and retrieval).

The outlined community blockchain interaction patterns were used for the analysis of six existing permissioned blockchain systems, in order to show how well these systems support the implementation of each pattern. The selected systems are Corda, Hyperledger Fabric, Hyperledger Sawtooth, Multichain, Quorum, and Tendermint. The analysis was performed on each pattern, classifying the possible solution complexity into an out of the box solution (OOTB), a customized solution (CS) and an architectural breaker (AB). The results of the analysis point out that the Transfer (P1), Exchange (P2) and Chat (P4) pattern are very well supported by most of the systems. For the implementation of these patterns, most systems provide extensive documentation, including code samples and example applications. Besides that, other patterns revealed unresolved issues and architectural limitations in existing permissioned blockchain systems. With the Election pattern (P5) it could be shown, that although the examined systems provide both read and write permissions, they lack of strong built-in mechanisms to ensure unlinkability. The Sharing pattern (P3) showed that time-based events are mostly not supported. The

Tracking pattern (P6) helped to point out deficiencies in the efficient data retrieval provided by the built-in querying mechanisms.

Furthermore, the universal applicability of the outlined community blockchain interaction patterns across use cases from different industries has been shown. Therefore, typical blockchain use cases were examined and discussed according to their requirements and matched with the patterns.

## 8.2 Future Work

Based on the results of this thesis, the next step is an analysis of more potential use cases in different industries. This can result in a comprehensive catalogue, that describes use cases, which consist of one or more community blockchain interaction patterns. Furthermore an extensive pattern-based analysis on other permissioned blockchain systems can be conducted, to evaluate other implementations according to their pattern support and solution complexity.

The use case catalogue in combination with an extended pattern-based system analysis can be used to create a new recommendation system. This could help interested companies, to quickly identify a suitable permissioned blockchain system, that provides the required components for a given use case.

# List of Figures

# List of Tables

# Bibliography


[AEVL16] A. Azaria, A. Ekblaw, T. Vieira, and A. Lippman. MedRec: Using Blockchain for Medical Data Access and Permission Management. In *2016 2nd International Conference on Open and Big Data (OBD)*, pages 25–30, Vienna, Aug 2016.

[Ant14] A. M. Antonopoulos. *Mastering Bitcoin: Unlocking Digital Crypto-Currencies*. O'Reilly Media, Inc., 1st edition, 2014.

[ARF+19] M. Andoni, V. Robu, D. Flynn, S. Abram, D. Geach, D. Jenkins, P. McCallum, and A. Peacock. Blockchain technology in the energy sector: A systematic review of challenges and opportunities. *Renewable and Sustainable Energy Reviews*, 100:143 – 174, 2019.

[Bac02] A. Back. Hashcash - A Denial of Service Counter-Measure. `http://www.hashcash.org/papers/hashcash.pdf`, 2002. Accessed: 20 May 2019.

[BCGH16] R. G. Brown, J. Carlyle, I. Grigg, and M. Hearn. Corda: An introduction. `https://docs.corda.net/_static/corda-introductory-whitepaper.pdf`, 2016. Accessed: 20 May 2019.

[Big18] BigchainDB 2.0 - The Blockchain Database. `https://www.bigchaindb.com/whitepaper/bigchaindb-whitepaper.pdf`, 2018. Accessed: 14 May 2019.

[BP17] M. Bartoletti and L. Pompianu. An Empirical Analysis of Smart Contracts: Platforms, Applications, and Design Patterns. In M. Brenner, K. Rohloff, J. Bonneau, A. Miller, P. Y.A. Ryan, V. Teague, A. Bracciali, M. Sala, F. Pintore, and M. Jakobsson, editors, *Financial Cryptography and Data Security*, pages 494–509, Cham, 2017. Springer International Publishing.

[Bro18] R. G. Brown. The corda platform: An introduction. `https://www.corda.net/content/corda-platform-whitepaper.pdf`, 2018. Accessed: 20 May 2019.

[Buc16] E. Buchman. Tendermint: Byzantine Fault Tolerance in the Age of Blockchains. Master's thesis, University of Guelph, Canada, 2016.

[But14] V. Buterin. Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform. https://github.com/ethereum/wiki/wiki/White-Paper, 2014. Accessed: 14 May 2019.

[But15] V. Buterin. On Public and Private Blockchains. https://blog.ethereum.org/2015/08/07/on-public-and-private-blockchains, 2015. Accessed: 14 May 2019.

[CDP19] F. Casino, T. K. Dasaklis, and C. Patsakis. A systematic literature review of blockchain-based applications: Current status, classification and open issues. *Telematics and Informatics*, 36:55 – 81, 2019.

[Dai98] W. Dai. B-money. http://www.weidai.com/bmoney.txt, 1998. Accessed: 20 May 2019.

[Ell91] C. Ellis. GROUPWARE: Overview and Perspectives. In W. Brauer and D. Hernández, editors, *Verteilte Künstliche Intelligenz und kooperatives Arbeiten*, pages 18–29, Berlin, Heidelberg, 1991. Springer Berlin Heidelberg.

[ET17] J. Eberhardt and S. Tai. On or Off the Blockchain? Insights on Off-Chaining Computation and Data. In F. De Paoli, S. Schulte, and E. Broch Johnsen, editors, *Service-Oriented and Cloud Computing*, pages 3–15, Cham, 2017. Springer International Publishing.

[Fab19] Hyperledger Fabric - Official Documentation. https://hyperledger-fabric.readthedocs.io/en/latest/, 2019. Accessed: 20 May 2019.

[GFM16] V. Garousi, M. Felderer, and M. V. Mäntylä. The need for multivocal literature reviews in software engineering: complementing systematic literature reviews with grey literature. In *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering*, EASE '16, pages 26:1–26:6, New York, NY, USA, 2016. ACM.

[GR18] J. Golosova and A. Romanovs. Overview of the Blockchain Technology Cases. In *2018 59th International Scientific Conference on Information Technology and Management Science of Riga Technical University (ITMS)*, pages 1–6, Riga, Oct 2018.

[Gre15] G. Greenspan. MultiChain Private Blockchain — White Paper. https://www.multichain.com/download/MultiChain-White-Paper.pdf, 2015. Accessed: 20 May 2019.

[Hea16] M. Hearn. Corda: A distributed ledger. https://www.corda.net/content/corda-technical-whitepaper.pdf, 2016. Accessed: 20 May 2019.

[KSHK18] K. N. Khaqqi, J. J. Sikorski, K. Hadinoto, and M. Kraft. Incorporating seller/buyer reputation-based system in blockchain-enabled emission trading application. *Applied Energy*, 209:8 – 19, 2018.

[LLX+18] Y. Liu, Q. Lu, X. Xu, L. Zhu, and H. Yao. Applying Design Patterns in Smart Contracts. In S. Chen, H. Wang, and L. Zhang, editors, *Blockchain – ICBC 2018*, pages 92–106, Cham, 2018. Springer International Publishing.

[LSP82] L. Lamport, R. Shostak, and M. Pease. The Byzantine Generals Problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, July 1982.

[Mer80] R. C. Merkle. Protocols for Public Key Cryptosystems. In *1980 IEEE Symposium on Security and Privacy*, pages 122–122, Oakland, CA, USA, April 1980.

[MKS10] R. Mordinyi, E. Kühn, and A. Schatten. Space-Based Architectures as Abstraction Layer for Distributed Business Applications. In *2010 International Conference on Complex, Intelligent and Software Intensive Systems*, pages 47–53, Feb 2010.

[ML17] A. Mavridou and A. Laszka. Designing Secure Ethereum Smart Contracts: A Finite State Machine Based Approach. *CoRR*, abs/1711.09327, 2017.

[MLI+18] F. Muratov, A. Lebedev, N. Iushkevich, B. Nasrulin, and M. Takemiya. YAC: BFT Consensus Algorithm for Blockchain. *CoRR*, abs/1809.00554, 2018.

[MVO96] A. J. Menezes, S. A. Vanstone, and P. C. Van Oorschot. *Handbook of Applied Cryptography*. CRC Press, Inc., Boca Raton, FL, USA, 1996.

[Nak08] S. Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System. `https://bitcoin.org/bitcoin.pdf`, 2008. Accessed: 14 May 2019.

[OBM+18] K. Olson, M. Bowman, J. Mitchell, S. Amundson, D. Middleton, and C. Montgomery. Sawtooth: An Introduction. `https://www.hyperledger.org/wp-content/uploads/2018/01/Hyperledger_Sawtooth_WhitePaper.pdf`, 2018. Accessed: 14 May 2019.

[OO14] D. Ongaro and J. Ousterhout. In Search of an Understandable Consensus Algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 305–320, Berkeley, CA, USA, 2014. USENIX Association.

[Pop18] S. Popov. The Tangle. `https://assets.ctfassets.net/r1dr6vzfxhev/2t4uxvsIqk0EUau6g2sw0g/45eae33637ca92f85dd9f4a3a218e1ec/iota1_4_3.pdf`, 2018. Accessed: 14 May 2019.

[Quo16] Quorum Whitepaper. https://github.com/jpmorganchase/quorum-docs/blob/master/Quorum%20Whitepaper%20v0.1.pdf, 2016. Accessed: 20 May 2019.

[Sch01] R. Schollmeier. A Definition of Peer-to-Peer Networking for the Classification of Peer-to-Peer Architectures and Applications. In *Proceedings First International Conference on Peer-to-Peer Computing*, pages 101–102, Linkoping, Sweden, Aug 2001.

[Swa15] M. Swan. *Blockchain: Blueprint for a New Economy.* O'Reilly Media, Inc., 1st edition, 2015.

[WG18] K. Wüst and A. Gervais. Do you need a Blockchain? In *2018 Crypto Valley Conference on Blockchain Technology (CVCBT)*, pages 45–54, June 2018.

[WS19] C. R. Worley and A. Skjellum. Opportunities, Challenges, and Future Extensions for Smart-Contract Design Patterns. In W. Abramowicz and A. Paschke, editors, *Business Information Systems Workshops*, pages 264–276, Cham, 2019. Springer International Publishing.

[WZ18] M. Wöhrer and U. Zdun. Design Patterns for Smart Contracts in the Ethereum Ecosystem. https://eprints.cs.univie.ac.at/5665/1/bare_conf.pdf, 2018. Accessed: 20 May 2019.

[XPZ+18] X. Xu, C. Pautasso, L. Zhu, Q. Lu, and I. Weber. A Pattern Collection for Blockchain-based Applications. In *Proceedings of the 23rd European Conference on Pattern Languages of Programs*, EuroPLoP '18, pages 3:1–3:20, New York, NY, USA, 2018. ACM.

[XWS+17] X. Xu, I. Weber, M. Staples, L. Zhu, J. Bosch, L. Bass, C. Pautasso, and P. Rimba. A Taxonomy of Blockchain-Based Systems for Architecture Design. In *2017 IEEE International Conference on Software Architecture (ICSA)*, pages 243–252, April 2017.

[ZWSL17] P. Zhang, J. White, D. C. Schmidt, and G. Lenz. Applying Software Patterns to Address Interoperability in Blockchain-based Healthcare Apps. *CoRR*, abs/1706.03700, 2017.

[ZXD+17] Z. Zheng, S. Xie, H. Dai, X. Chen, and H. Wang. An Overview of Blockchain Technology: Architecture, Consensus, and Future Trends. In *2017 IEEE International Congress on Big Data (BigData Congress)*, pages 557–564, June 2017.