KOSMoS Private Blockchain Toolkit: How to Use Hyperledger in an Industrial DLT Project

Martin Schäffner[1], Constantin Lichti[2], Jonas Gross[3], Philipp Sandner[4]

Associated research project:

Kollaborative Smart-Contracting-Platform für digitale Wertschöpfungsnetze (KOSMoS)

Funding authority: German Federal Ministry of Education and Research (BMBF)

Funding code: 02P17D020

Publication date: March 2021

Version: 1.0 (Work-in-Progress)

[1] Datarella GmbH, Germany.
[2] Frankfurt School Blockchain Center at the Frankfurt School of Finance & Management GmbH, Germany, Corresponding author, constantin.lichti@fs-blockchain.de.
[3] Frankfurt School Blockchain Center at the Frankfurt School of Finance & Management GmbH, Germany.
[4] Frankfurt School Blockchain Center at the Frankfurt School of Finance & Management GmbH, Germany.

Preface

This paper addresses the potential users and developers of a DLT-based platform building on the industrial KOSMoS project and clarifies the fundamental role of distributed ledger technology (DLT) within the KOSMoS system. Based on the description of the main use cases, a basic understanding about the applications of the system and in-depth insights into the implementation of Hyperledger are provided. Finally, related DLT-based business models complete the content of the KOSMoS private blockchain toolkit. As we continue to work on implementing the Hyperledger-based industrial use cases, the document's content is subject to change. A final version will be published after the project is finished.

Table of contents

Glossary

BFT	Byzantine Fault Tolerance (BFT) is the feature of a distributed network to reach consensus (agreement on the same value) even when some of the nodes in the network fail to respond or respond with incorrect information.
Certificate Authority	The Certificate Authority (CA) provides a number of certificate services to users of a Hyperledger Fabric system.
Consensus mechanism	A consensus mechanism is a fault-tolerant mechanism that is used in computer and blockchain systems to achieve the necessary agreement on a single data value or a single state of the network among distributed processes or multi-agent systems, such as with cryptocurrencies.
Docker	Docker is an open-source software platform to create, deploy and manage virtualized application containers on a common operating system (OS) with an ecosystem of allied tools.
Fabric SDK	The Hyperledger Fabric SDK allows applications to interact with a Fabric blockchain network. It provides a simple API to submit transactions to a ledger or query the contents of a ledger with minimal code.
LDAP	The Lightweight Directory Access Protocol (LDAP) is an open, vendor-neutral, industry standard application protocol for accessing and maintaining distributed directory information services over an Internet Protocol (IP) network.
Membership Service Provider (MSP)	A MSP is a Hyperledger Fabric component that offers an abstraction of membership operations like issuing certificates, validating certificates, and user authentication.
Peer	Peers host ledgers and smart contracts and verify transactions.

Raft-BFT	Raft is a consensus algorithm designed as an alternative to the Paxos family of algorithms.
Smart contract	A smart contract is a piece of code enabling interaction with the network’s shared ledger. Also called chaincode.
X.509 certificates	An X.509 certificate is a digital certificate that uses the widely accepted international X.509 public key infrastructure (PKI) standard to verify that a public key belongs to the user, computer or service identity contained within the certificate.

List of figures

1. Target group and scope of the document

Overview. This document has been written within the scope of the KOSMoS research project funded by the German Federal Ministry of Education and Research (BMBF) under the funding code 02P17D020. The research project seeks to develop a collaborative platform for small and medium-sized companies (SMEs) in the manufacturing sector based on distributed ledger technology (DLT). DLT is a family of innovative, decentralized technologies with blockchain technology being the most popular member of the family. Blockchain technology, in a narrow sense, serves as a technological basis of the crypto assets like Bitcoin, but can also be applied for use cases outside the payment sector. Within the KOSMoS project, two different industrial use cases are realized on a DLT-based platform: 1) dynamic leasing, and 2) transparent maintenance (more details in Chapter 2.1).

Goals. This document has various goals. First, this toolkit discusses the **benefits of using DLT in the context of the manufacturing industry** and shows how DLT-based systems can be realized to improve business processes and increase efficiency. We share our research results obtained within the KOSMoS project to the general public and, in particular, to industrial companies. Despite the enormous potential of DLT, to date, the use of DLT in the German and European industry is not widespread. This is also due to the fact that the benefits of using DLT are not well recognized within the industry. This document seeks to educate in this regard.

Second, it is explained in detail **how a DLT-based system can be set up**. For the KOSMoS project, it was decided to use **Hyperledger Fabric** as the underlying DLT system (for more details about the choice of the technology, see Gross et al., 2020a). This toolkit guides how such a Hyperledger DLT platform can be set up in the context of the industry, aiming to lower the threshold of market entry for DLT-interested companies.

Target group. Building this toolkit, we intend to address the possible needs of players in the manufacturing industry, especially for cross-company business models and Industry 4.0 solution approaches. **Manufacturers, suppliers, and third parties** can get the guidance, how and in which context to use DLT to increase the efficiency of current business processes and possibly to implement **new cross-company business models**, e.g., pay-per-use and automation business models. **Customers** can decrease their leasing fees as leasing fees are calculated by a smart contract in the DLT network, and transparency is heavily increased. **Developers** get in-depths insights into **blockchain-based implementation codes** of real-world industrial use cases.

Structure. The toolkit is structured as follows. In Chapter 2, the industrial **use cases are explained**, which were implemented in Hyperledger. Further, the benefits of using DLT and the concrete **choice of Hyperledger for the KOSMoS system is explained** in detail. Chapter 3 shows **how the two KOSMoS use cases can be implemented in Hyperledger** and which technological requirements need to be fulfilled. In Chapter 4, we **discuss related use cases** of DLT that could be realized in the Industry 4.0 and supply chain sector but were not within the scope of this research project.

State. As we continue to work on implementing the DLT-based use cases, the document's content – especially regarding Chapter 3 – is still **subject to change**. Changes made in the document will be published in a final version after the project is finished.

2. The KOSMoS system: Industrial use cases implemented in Hyperledger

2.1. The KOSMoS research project

Our goal. The KOSMoS research project aims to develop **a blockchain-based platform for cross-company collaboration of production and processing data** offering new services and business models that can be integrated into existing systems of the involved parties. For the purpose of this project, we investigate two use cases:

- **dynamic leasing**, and
- **transparent maintenance**.

Implementing these business models aims at offering benefits for all cooperating companies by achieving efficiency gains, e.g., via lowering prices, maintenance costs, or easier product distribution. Smart contracts play a significant role in order to achieve this efficiency gain. Due to the elimination of trusted third parties, and a transparent and cryptographically secure platform, data can be sent and processed in a trusted manner to improve efficiency. In summary, we strive to establish a novel form of cross-company business models that leverages the advantages of DLT.

Our approach. In order to be able to **share confidential data** securely across company borders, some requirements must be met. In addition to the secure and authentic transmission of data, it must be given that the information is stored securely against manipulation. For this purpose, the KOSMoS platform uses the DLT Hyperledger Fabric. By chaining data entries

cryptographically together, subsequent manipulation of the data is not possible. In order to transfer the recorded data safely and unchanged into Hyperledger, a transaction is created and signed as close as possible to the creation of sensor- or other ledger-relevant data. Thus, it will be ensured directly during data acquisition that information can no longer be changed or manipulated.

Once the data has entered the blockchain, **smart contracts are used** to process the data. Smart contracts in Hyperledger, also called **chaincode**[5], receive and automatically execute actions if pre-specified conditions are met. For example, **a leasing fee can be adjusted accordingly and pre-determined in the chaincode based on the usage of a machine or the performed maintenance tasks**. The system architecture is further enhanced by additional components such as an edge device, an analysis platform, and the KOSMoS user interface. Combined, this results in a reusable framework that can be adopted, in particular by SMEs, to implement business models within partner companies with only a fair amount of effort.

2.2. Use cases

In the following, two major use cases of DLT in the manufacturing industry investigated within this project are explained: **dynamic leasing** (2.2.1) and **transparent maintenance** (2.2.2). This Chapter focuses on describing the functionality as well as the various advantages of these use cases in detail. For both use cases, the structure is as follows: First, the use cases are roughly described, followed by the problems of the current states. Second, important stakeholders are identified. Finally, the goals of the use cases are described in detail.

2.2.1. Dynamic leasing

This use case aims to offer the possibility to perform dynamic leasing with a leasing fee derived from the actual use and wear from the machine. At the current state, customers can either purchase the machines from the industrial partner directly or use them as part of a lease with a static fee. As part of the KOSMoS project, the leasing is converted into a leasing model with a dynamic leasing fee. The leasing rates are calculated depending on the machine use (duration, tool change, crash, etc.) and maintenance (digital maintenance plan with rules and regulations, etc.). The customer will be able to see the parameters relevant to the leasing via a real-time dashboard. In addition, the customer will receive a forecast of the development of

[5] In the context of Hyperledger Fabric, we use the terms smart contract and chaincode interchangeably.

his leasing rate based on historical and current usage. Each data that affects the leasing fee is documented in the blockchain and processed via smart contracts.

Status quo. The leasing is currently carried out via static monthly leasing contracts for the machine tools while the fees are fixed at the beginning of the leasing agreement. Oftentimes, a leasing provider is involved, who manages the leasing, which incurs additional costs. This usually results in high costs that are not related to the use of the machine. In addition, machine users are aiming for a higher share of dynamic leasing. At the moment, the rate of dynamic leasing contracts is still low. Dynamic leasing is also attractive for machine manufacturers since it can increase customer loyalty.

From a technical view, **most components for dynamic leasing already exist.** The machine is equipped with lots of sensors that produce data which helps to create cost factors for leasing fees.

Stakeholders. At this point, we assume there are two parties involved – the industrial partner and a customer. However, it is not excluded that other parties can be added to the system. The machine manufacturer is the owner of the machine. Leveraging dynamic leasing and the documentation of sensor and production data, the manufacturer gets a better overview of how the machine is used. This makes the collaboration more flexible. At the end of the lease, the machine is often returned to the machine manufacturer. Dynamic leasing makes it possible to document the use and wear of the machine so that the current value can be determined much more precisely. The user of the machine now has significantly more control over the leasing fee. If the machine is used more intensively, the leasing amount increases but reduced if the machine is used less. It is also possible to see the current leasing fee in real-time.

Goals. The aim of this use case is a **dynamic calculation and billing of the leasing fee based on the documentation of machine use and machine wear using smart contracts.** To achieve this, the sensor data must be converted into a format (transaction) that the chaincode can understand. The data may be pre-processed to keep the amount of data relatively small and manageable.

This documentation **increases disintermediated data control by transparency** on the one hand and paves the way for novel and efficient pay-per-use business models on the other. Both parties can rely on the authenticity and integrity of the data because they are stored on the blockchain. The leasing rate and the current use should be made available to the user via an intuitive frontend. The machine manufacturer also has the advantage that the value after the lease can be determined and justified more easily and clearly.

2.2.2. Transparent maintenance

To prevent machine downtimes, machine builders usually publish maintenance instructions for each specific part of the machine. **On the one hand, it is not ideal that the maintenance instructions are designed for the average machine usage without taking actual machine usage into account**. On the other hand, due to lack of support, maintenance intervals are often not adhered to (e.g., the maintenance instructions are currently only available on paper). In the case of heavy machine stress, this usually leads to machine downtimes, which are, however, avoidable.

As part of this project, the maintenance instructions should be digitized to actively assist machine operators with relevant maintenance data. Due to the availability of the information on the machine stress in the platform, the **maintenance instructions can be automatically adapted to the usage**. In case of a damage despite maintenance measures, the transparency of the maintenance clearly defines the responsibility for the repair and could give an undoubtable indication how the damage might have occurred. However, most importantly is the documentation of the performed maintenance. This gives future maintainers a clear and precise overview of how the machine was maintained in the past, which in turn enables them to perform the maintenance more targeted and more efficiently as they do not need to wait for paper-based maintenance logs from earlier measures.

Status quo. Machines that have been delivered to customers are usually serviced by third party customers. However, the maintenance staff of the industrial partner must also perform maintenance in some cases. If this is the case, **it is often difficult to recreate previous maintenance measures, which were often not fully documented**. In addition, there is hardly any information about the intensity of use and previous error messages from the machine. This carries the risk that maintenance cannot be carried out optimally and the actual causes of error messages remain undetected. Furthermore, existing maintenance instructions are often only available in analog form on paper. Before this use case can be implemented, the industrial partner must be able to provide the maintenance plans in digital forms. Based on these maintenance plans, the maintenance log can then be designed and implemented.

Stakeholders. There are essentially three main stakeholders. On the one hand the machine manufacturer (also called industrial partner) and on the other hand the customer. Additionally, there can be a third party company that performs the actual maintenance.

In a few cases, the machine manufacturer carries out maintenance at the customer's machines. For the delivered machines, there are often limits to the loading and utilization of the machine, in which the manufacturer guarantees a largely smooth operation. Therefore,

there is great interest in the manufacturer that the customer does not use the machine beyond pre-defined limits.

The machine manufacturer's customer operates and maintains the machines based on specified maintenance measures. These maintenance measures are documented during or after the implementation so that they can be viewed later.

Goals. The aim of the use case is to provide a **blockchain-based maintenance log**, where all involved parties are informed in real-time about maintenance measures and error messages from the machine. The use of blockchain technology ensures that data is protected against manipulation. In case damage occurs at the machines, the maintenance log can help finding the problem from a trusted and non-tampered data stored on the blockchain.

2.3. The role of DLT within the KOSMoS project

The KOSMoS system. The Hyperledger Fabric DLT used in this project is part of the following logical system overview (see Fig. 1) where most of the individual components are accessible over standardized ports, others are only available in private networks.

Definition of a blockchain. Technically, a blockchain refers to a cryptographically secured distributed ledger that uses a decentralized consensus mechanism to consistently order and validate transactions across multiple peers (Glaser et al., 2019). Blockchains are the most prominent type of DLTs, in which transactions of a network are stored decentralized on many peers but which could apply different consensus mechanisms. *Public* and *permissionless* blockchains such as Bitcoin and Ethereum allow every internet user to access decentralized applications and participate in core functions. For instance, the core functionality of the Bitcoin platform is the usage of Bitcoin tokens (BTC) as a universal means of payments. In contrast, *private* blockchains such as Hyperledger Fabric are centrally maintained by a single entity or small group of organizations.

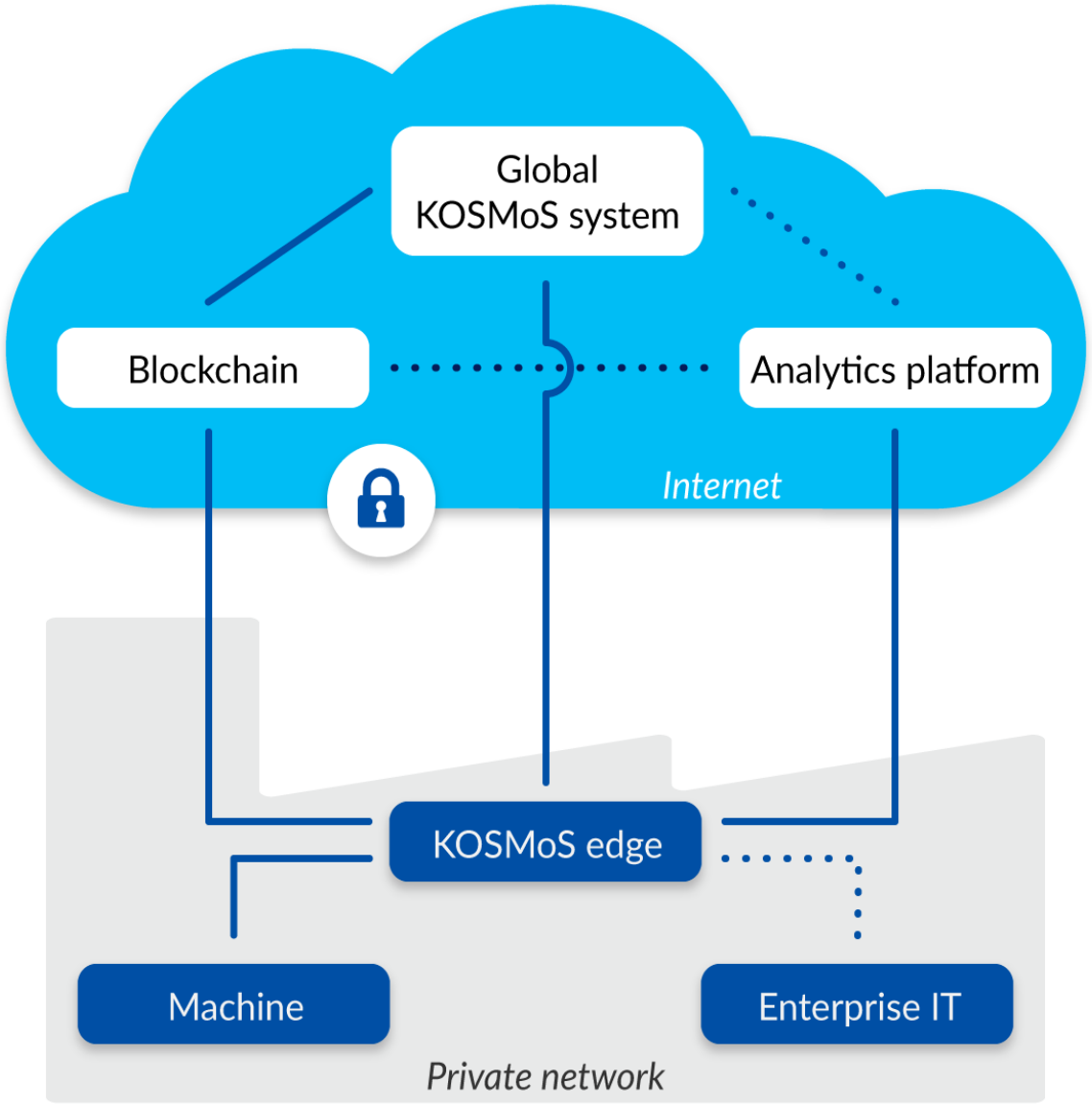

Figure 1: Overview of the KOSMoS system including logical system components

Figure 2 shows the simplified components of a peer. The peers contain information for each individual channel such as the data storage of the relevant machine and production data, the chaincode that handles incoming transactions and performs operations on the ledger, as well as identity-management which do not necessarily need to be managed by the peer, but rather access capabilities from additional services, like keycloak.

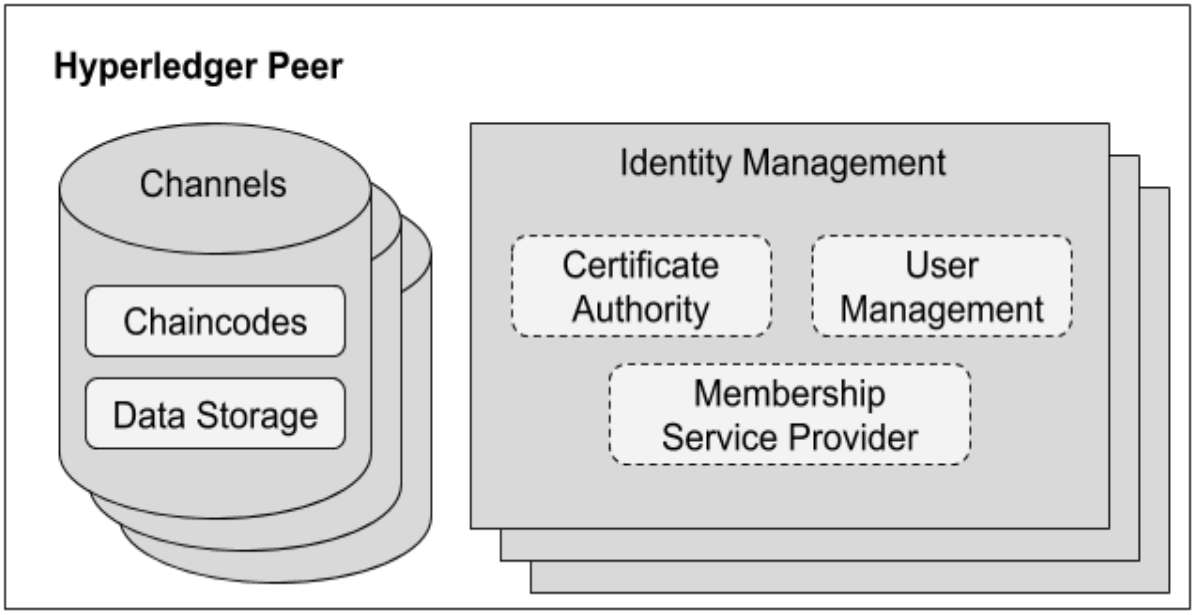

Figure 2: Hyperledger peer component of the KOSMoS system

The role of smart contracts. From an application developer's point of view, a smart contract together with the ledger forms the core of a Hyperledger Fabric blockchain system. While a ledger contains data about the current and historical status of a number of business objects, a smart contract defines the executable logic that generates new data that is added to the ledger.

A contract from the user's point of view represents business procedures, responsibilities, and payment terms. These include, for example, the term, the contracting parties, and the subject matter of the contract such as how the leasing fee is calculated. A smart contract represents this contract in the Hyperledger system. The contract is divided into small subcategories. For example, there are several smart contracts for the use case of dynamic leasing - each for the creation and administration of a machine, a tariff, the calculation logic of a leasing fee, the production data and a contract, which in this sense stores and references the general data such as term, machine, and tariff. The modular structure allows dynamic adaptation of the elements of the contract. Thus, in case of a change in the cost factors, only the tariff and the contract that is to reference the new tariff needs to be adapted.

Why using a blockchain for this project? A blockchain ensures the correctness of information to be verified by a consensus mechanism rather than by a central instance. Depending on the application and the respective use case, the Hyperledger blockchain in KOSMoS stores different information and therefore serves different purposes:

1) Integrity check of sensor- and production data

Once data is created from the sensors and from the machine itself, we need to ensure that this **data is not able to be manipulated**. The transaction metadata and payload will be hashed to create a unique value for the contents inside the block. This value is linked to the following block, resulting in a traceable chain of information that is linked together – a blockchain. The blockchain consensus and verification mechanism avoids manipulation by design. It ensures that once a transaction is about to be published to the blockchain, the inputs are simulated to make sure there is no conflict in the system. If there is a conflict detected, the transaction fails at the simulation part and will not be published to the ledger, hence preserving the data integrity of the system. **Checking data integrity is relevant across all use cases.**

In addition to the actual integrity of a data in blocks, it is also possible to derive and add a hash value from the data and add it to the set of data written on the ledger. The data set keeps the hash from the data of the previous block and creates a new hash that serves as the basis for the following data. This is called a hash-chain. Visualizing this could raise awareness about how the blockchain is used and increases the barrier for manipulation.

2) Immutable data storage

In this case, the **relevant data is stored directly on the blockchain.** This variant is only suitable for small amounts of data, like the accumulated number of produced pieces, since the scaling and speed of a blockchain-based database cannot keep up with the vast sizes of data since the data has to be put in transactions that need to be signed, verified, and published. Even though this could be done within seconds, it disqualifies for big datasets. An example in the KOSMoS project is the storage of the maintenance logbook (see 2.2.2 Transparent maintenance). Only small amounts of data, which can be stored directly on the blockchain, are sufficient to indicate maintenance *to be carried out* and to confirm that maintenance *has been carried out*. In this way, the true maintenance history becomes transparent at any time.

3) Authorization

In both of the above-mentioned possibilities, the KOSMoS system requires (limited) access to the stored information. To ensure that every party is only allowed to see the information to which she has access, a channel-based blockchain is used in the KOSMoS system (i.e., Hyperledger). For each connection between organizations a separate channel is opened, based on “sub-blockchains”.

In addition to the recorded information, chaincodes are also stored and executed on-chain. Chaincode is software that can be invoked by specific types of transactions. In the KOSMoS

case, **part of the business logic is implemented in the chaincode**, which is used to process incoming data, e.g., for dynamic leasing. For example, a chaincode could calculate the respective leasing fee to be paid for month X from cost factor information that is provided in another chaincode linked to the tariff. In other words, a chaincode contains an unchangeable and predefined execution logic of an application case. An execution environment for chaincode is included in the blockchain implementation. The chaincodes are being deployed locally on each peer and are accessible within the entire channel/network.

Note that a chaincode cannot replace a normal contract. There is still a contract required that defines the processing logic and its consequences. A chaincode only defines how incoming data will be processed.

2.4. Introduction to Hyperledger: Concept, structure and network overview

In the following, we provide an overview of the chosen DLT platform Hyperledger Fabric including its network parties and the transaction flow. Furthermore, a comparison of different DLT platforms is shown to explain Hyperledger Fabric's suitability for the KOSMoS project.

2.4.1. Overview

Hyperledger Fabric is a modular blockchain framework that acts as a foundation for developing blockchain-based products, solutions, and applications using plug-and-play components that are aimed for use within private enterprises. Traditional blockchain networks cannot support private transactions and confidential contracts that are of utmost importance for businesses. Hyperledger Fabric was designed in response to this as a modular, scalable, and secure foundation for offering industrial blockchain solutions.

Hyperledger Fabric is the open-source engine for blockchain and takes care of the most important features for evaluating and using blockchain for business use cases. Within private industrial networks, the verifiable identity of a participant is a primary requirement. Hyperledger Fabric supports memberships based on permission; all network participants must have known identities. Many business sectors, such as healthcare and finance, are bound by data protection regulations that mandate maintaining data about the various participants and their respective access to various data points. Fabric supports such permission-based membership.

Hyperledger aggregates various infrastructures and projects based on DLTs developed by the Linux Foundation, where Hyperledger Fabric is one specific project. Hyperledger Fabric operates on an access-restricted, also called **permissioned**, ledger. The architecture of Hyperledger Fabric is based on multiple ledgers operating independently of each other. However, it allows a transaction of one ledger to view and address the transactions and chaincodes of another ledger. Hyperledger Fabric provides an extensible and modular architecture that can be used for different areas and is, therefore, independent of a specific field of application.

The access restriction of Fabric, as well as the validating and non-validating peers, are defined by the network operator(s). The network operator can allocate different access rights to users in order to carry out the necessary transactions within the network. Hyperledger Fabric's restricted character stems from **users' need for privacy.** However, privacy does not apply to regulatory authorities, which must have the possibility of identification and verification. The encryption of the identity, for example, could therefore be carried out in such a way that it remains hidden from other unwanted participants but can be viewed, for example, by regulators. However, since there is an actual business operated on the Hyperledger system, it is common to use human-readable identifiers.

Within Hyperledger Fabric, content confidentiality is achieved by encrypting transactions so that only those involved can decrypt and execute them. Transactions between parties are conducted in "channels": Certain users communicate in a sub-network (sub-channels) to which only authorized users have access.

2.4.2. The Hyperledger network

In the following, the different parties involved in the Hyperledger network are explained (see Fig. 3).

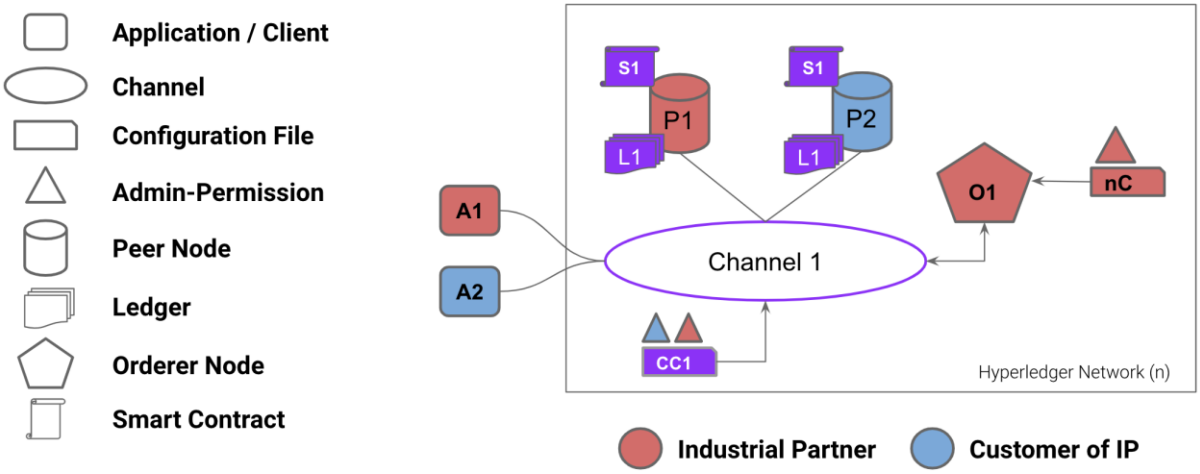

Figure 3: Overview of the Hyperledger network

- **Peer nodes:** Store the actual ledger and deployed chaincodes
    a. Endorser: Prevent those non-verified transactions to be distributed in the network
    b. Committer: Add validated transactions to channel-specific ledgers
- **Orderer nodes:** Collect transactions, aggregate them into blocks and can work across various channels
- **The certificate authority (CA):**
    a. Introduces new channels and creates the first block (genesis block)
    b. Registers identities of organizations, peers, and clients via public keys
    c. Manages the rights assigned to identities via certificates
    d. Creates Membership Services Providers (MSPs) collecting access rights

2.4.3. Transaction flow

The process of creating, signing, submitting, and validating a transaction in Hyperledger is a process that involves a set of actors that perform different operations. The required steps are listed below:

1. **Client application:** Initiates transaction proposal (Tp) with the intention to update the ledger, signs the Tp, and distributes the transaction to the endorser node
2. **Endorser node:**
    a. Validates incoming transaction with respect to format, potential double-spending or – with the assistance of a MSP – if the signature is valid
    b. Simulates by adding the read-write sets
    c. Signs transactions and sends it back to the application

3. **Client application:**
    a. Collects a sufficient amount of transaction proposals according to the endorsement policy
    b. Signs the transaction
    c. Sends to the orderer node
4. **Orderer node:**
    a. Receives transactions and sorts transactions in chronological order in the respective channel
    b. Aggregates transaction to blocks (no validation)
    c. Sends the block to the channel-specific committing-peers
5. **Committing peers:**
    a. Receive new blocks with transactions from the orderer node
    b. Compare the read-write set of the new blocks with the world state and verifies
    c. Updates world state with new transactions

2.4.4 Advantages and disadvantages of Hyperledger

A comparison of Hyperledger and other DLTs such as Ethereum, R3 Corda, and Stellar is shown in Figure 4 (for more information about the selection process of a blockchain platform see Gross et al., 2020a).

Hyperledger is exceptionally user-friendly, relatively easy to install, and can rely on a huge developer base. Further, a solid performance, high-cost efficiency, and safety are provided. We selected the Hyperledger Fabric protocol for the KOSMoS project for the following main reason: Hyperledger Fabric is very suitable for consortial blockchains as it is most appropriate for multi-client capability. It describes that multiple participants can operate on the blockchain system while transactions are only visible to relevant stakeholders. Therefore, Hyperledger Fabric incorporates channel functionalities. As we intend to use only one single blockchain for different use cases and, therefore, with different and independent parties, we need to make sure that the privacy concerns of the industrial partners are being respected. In Hyperledger Fabric, it is possible to isolate connections between organizations over channels in such a way that the transactions are hidden from other participants on the blockchain. This possibility protects data from unintended sharing with others.

	User friendliness		*Performance*	*Cost efficiency*		*Release capability*		*Security*	*Administration*
	Installation	Documentation	Oracles	Transaction fees	Maintenance costs	Community / developer	Maturity of DLT	System integrity	Interoperability
Ethereum	Available on GitHub	Very well documented on GitHub	Available	Gas price of 0 feasible	Large amount of external developers available	Very large	Sufficiently tested	Partly tested – DAO hack	Possible, but only with high effort
Hyperledger Fabric	Available on GitHub	Very well documented on GitHub	Available via Oraclize API	Not restricted to specify	Large amount of external developers available	Large	Only some minor tests	Not sufficiently tested	APIs available
R3 Corda	Available on GitHub	Shortly available on GitHub	Available	Not restricted to specify	Amount of external developers unclear	Medium	Only some minor tests	Not sufficiently tested	APIs available
Quasar / Stellar	Not available on GitHub – Docker images exist	Source code via GitLab exchangeable	Not clear	Not restricted to specify	Amount of external developers unclear	Small core developer team	Only some minor tests	Not sufficiently tested	APIs available

Figure 4: Comparison of various DLT platforms

Additionally, Hyperledger comes with a modular system architecture that allows developers to add other modules, like an identity module, easily into the system. Another advantage of Hyperledger is that the rights of each participant can be easily managed with so-called “Membership Service Providers” (MSP). Here, the certificate authority can create, sign, deposit, and revoke X.509 certificates, which grants the individual participants certain rights in the network. Further, the MSP can be connected to the organization’s identity management system.

Last, Hyperledger is very scalable and high performing. Any consensus mechanism can be implemented that allows multiple participants to send transactions that will be automatically ordered by the orderer node. In this project, we decided to implement the pBFT-similar BFT Raft consensus mechanism as it fits best the requirements, we need to fulfill within both use cases (for more information about the selection process of a consensus mechanism see Gross et al., 2020b).

3. Hyperledger applied: Industrial use cases and chaincode examples

This section focuses on the steps and processes required to implement Hyperledger-based industrial use cases. In particular, it is shown how the system architecture looks like, how the system is being governed, and how the interaction with and the processes inside the Hyperledger system look like.

3.1. General Hyperledger architecture and operations

The KOSMoS system architecture consists of multiple components. There is an architecture for the systems of the machines, a management tool, called the contract engine, an analysis system, and the blockchain system built on Hyperledger Fabric. This Chapter provides an overview over these systems that form the system architecture as well as a detailed explanation of the Hyperledger architecture and a hands-on guideline to set up the dynamic leasing.

3.1.1. KOSMoS system architecture

The system architecture of the use cases in the KOSMoS system comprises multiple components. Note that the components are different from systems. Components are bound to a specific purpose whereas systems fulfill specific tasks. As a result, multiple components integrate the same system, e.g., the Blockchain Connector. The overall system architecture is split up in five main components (see Fig. 5):

1. **KOSMoS Global**: This system comprises what every involved organization in the system can see. Machine manufacturers and their customers look at the same data. Note that KOSMoS Global includes the peers which hold the ledger of the KOSMoS Hyperledger system. Every peer that operates transactions is logically located in KOSMoS Global, for example, orderer nodes or peer nodes. Further, this is where the web application of KOSMoS will be located.

2. **KOSMoS Local**: This component comprises multiple subcomponents. In general, KOSMoS Local describes the applications and systems on the physical location or infrastructure of the client, such as the client's network or machines.

3. **Machine Edge**: This subsystem is the bridge between the machine gateway and KOSMoS Local or Global. It is responsible for handling and processing the data

correctly and distributing it to the other subsystems. The edge is the place where the client software is located to propose and create a transaction for Hyperledger Fabric. The edge communicates with the peer and orderer nodes from KOSMoS Global. Since the edge is located on the client's shop floor, this system is also included in KOSMoS Local.

4. **Machine gateway**: This subsystem represents the machine that acts as the source of data from its sensors or metadata from the production. This subsystem is located on the customer's shop floor and is also considered part of KOSMoS Local.

5. **Analysis Platform**: The fourth subsystem is the analysis platform. Its task is to process incoming data from the machine gateway and the edge to analyze, detect patterns, and to give recommendations for optimal operation of the machines.

The system architecture is shown in Figure 5.

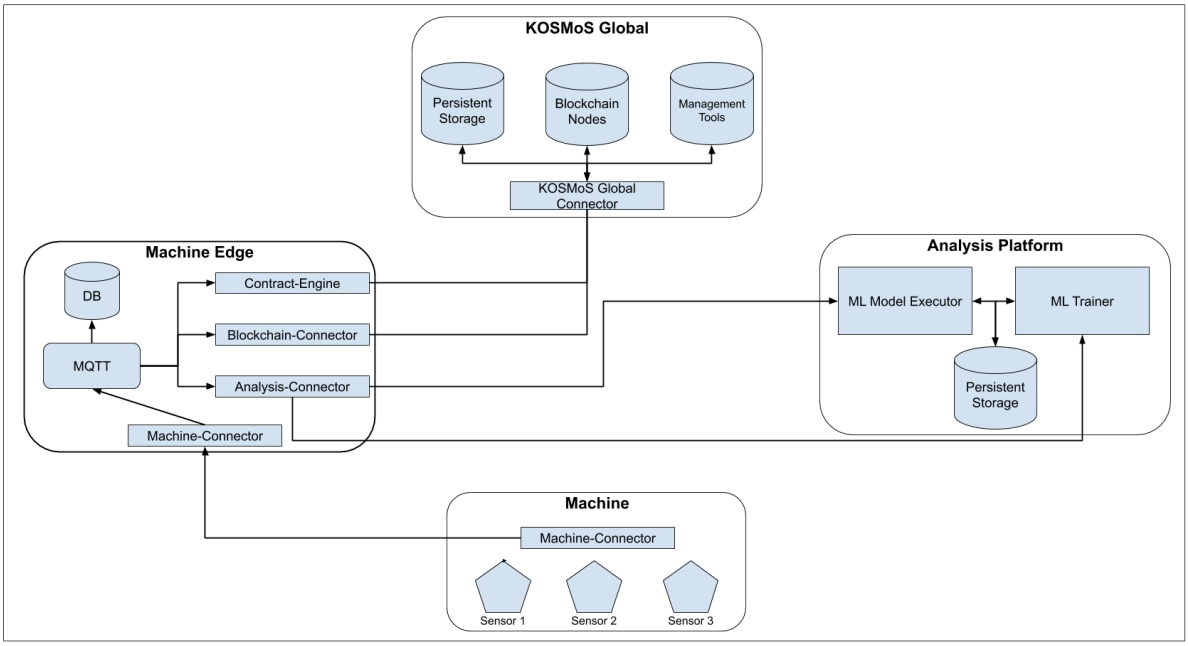

Figure 5: KOSMoS system architecture

3.1.2. Hyperledger architecture

Hyperledger Fabric exists at the highest level in the form of the Hyperledger network. This network can be operated and managed by one or more parties. The network itself can be used as a DLT solution since it already comprises an own ledger within a channel that is available to every actor. The great advantage of Hyperledger Fabric is, however, that so-called "channels" can be created within a network. Channels are sub-blockchains that exist within a

network and are independent of other channels. These channels are particularly suitable for the exchange of data between parties that should be protected from access by unauthorized parties. Only those who have access rights to the channel can see the content of the blockchain. Thus, channels can be used to implement data privacy features. Therefore, the use of channels between the industrial partner and its customers is particularly suitable.

The following figure (Fig. 6) shows a simplified overview of the actors and elements within Hyperledger Fabric on the example of creating and adding a transaction to the ledger.

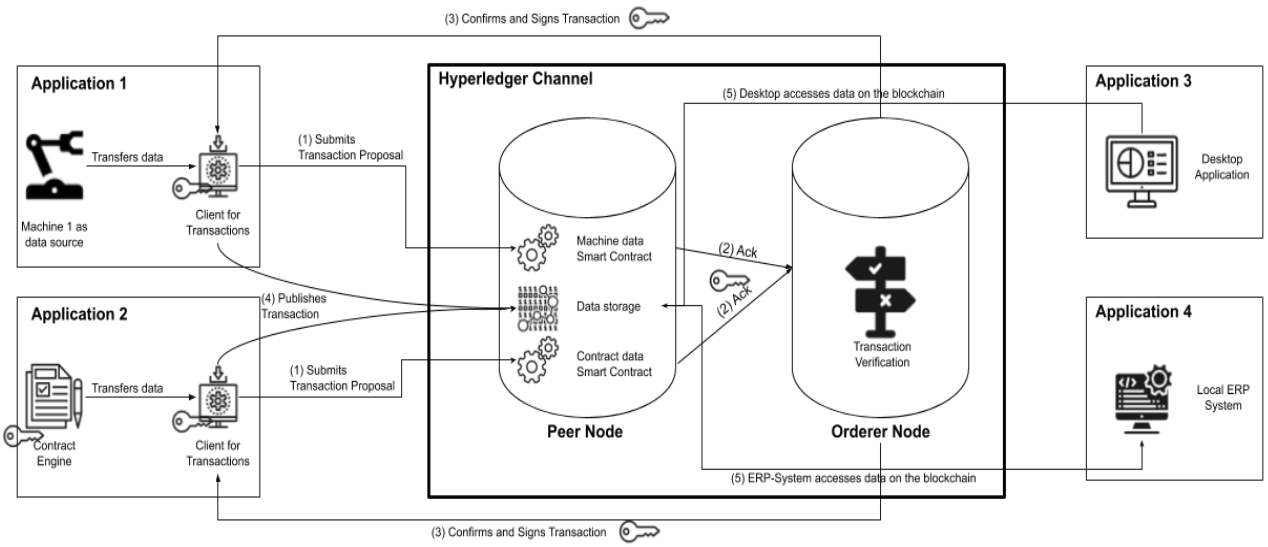

Figure 6: Actors and elements within Hyperledger Fabric

3.2. Hyperledger components

The previous Chapter introduced the Hyperledger architecture. This Chapter will take a closer look at the specific components of the Hyperledger architecture.

3.2.1. Hyperledger channel

A Hyperledger Fabric channel is a private “subnet” of communication between two or more specific network members in order to carry out private and confidential transactions. A channel is defined by members (organizations), peers per member, the shared ledger, chaincode-based applications, and the orderer nodes. Every transaction in the network is carried out on a channel on which each party must be authenticated and authorized to carry out transactions on this channel. Each peer who joins a channel has its own identity, which is assigned by a Membership Service Provider that authenticates each peer against its channel peers and services.

3.2.2. Peer nodes

Peers describe a type of server that participates in the consensus mechanism and the processing of transactions. There are different roles of peers. However, they can all run on the same machine and are more meant to be a logical division.

Endorsing peers

Endorsing peers are peers that simulate transactions in isolated chaincode containers and execute transaction proposals and generate proposal responses based on the chaincode results. The chaincode must be installed on all supporting peers.

Committing peers

These are the peers on which chaincode is not necessarily installed, but which hold the complete ledger of the data records. The main difference between committing and endorsing peers is that committing peers cannot call chaincode or execute chaincodes and includes both read and write functions.

Ordering nodes

Ordering nodes are special types of peers whose job is to receive endorsements from the endorsing peers, validate if the consensus is given, and send commands to the committing peers to update the ledger accordingly. Ordering nodes keep track of all transactions in their ledger, including valid and invalid transactions, while peering confirmation and commit contain only valid transactions.

Authorized and unauthorized Peers:

Data protection can be further refined using these two types of peers. While authorized peers keep the complete data in the ledger, unauthorized peers hold only part of the data, e.g., the block header. With the help of unauthorized peers, manipulation security can be increased without having to disclose the actual data.

3.2.3. Applications

An application can interact with a blockchain network by sending transactions to a ledger or querying ledger content. It uses functions from the Fabric SDK. An application must do six basic steps to complete a transaction:

1. Choose an identity from a wallet
2. Connect to a gateway

3. Establish access to the network
4. Create a transaction request
5. Submit transaction
6. Process response

The application accesses the contents of the ledger via the APIs provided and can display them from now on.

3.2.4. Certificate Authorities (CA)

Hyperledger Fabric networks fall into the category of permissioned blockchains. In order for the system to be able to validate whether a particular actor in the network is authorized to do what it intends to do, the system must be able to distinguish between actors. In Hyperledger Fabric, this is achieved using digital X.509 certificates.

A Certificate Authority (CA) is the component that is responsible for issuing digital certificates and signing them with their public key. In Fabric, these certificates bind an actor's public key to the actor himself. So if an actor A trusts a particular certification authority and has access to the public key of the certification authority, he can in turn trust another actor B who claims that it is B, by using a certificate issued by the certification body.

There is an integrated component in Fabric called Fabric CA, a private root certificate authority that can be used to create and manage digital identities in Fabric networks in the form of X.509 certificates. Hyperledger Fabric also offers the alternative to implementing existing LDAP systems from the players.

3.2.5. Membership Service Provider (MSP)

Membership Service Providers (MPS) store public keys of different actors. In short, MSPs turn identities into roles, which in turn determine whether an actor's permissions are at different levels. Simply put, MSPs are just a set of folders that are included in the network configuration and used to define organizational roles. In addition, they define which certification bodies are allowed to generate (valid) identities.

There are two types of MSPs: local MSPs and channel MSPs. Both work the same way, they transform identities into roles, but their scope differs. Local MSPs are defined for peers and orderers. A local MSP must be defined for each peer in a fabric network. This must be done to know who has permission to this peer, e.g., which members are allowed to contribute chaincode.

Similar to how local MSPs define who has authority at the peer level, channel MSPs define who has authority at the channel level. An MSP must be defined for every organization participating in a channel. This is very important because it defines which members of this organization can act on behalf of the organization. Channel MSPs are logically defined once in the channel configuration, but physically a channel MSP is also instantiated in the file system of each peer in the channel and synchronized via consensus.

3.2.6. Governance model

The purpose of a governance model is to give a closed system a structure that all participants accept and that should be adhered to. The needs and obligations of the actors in the system are precisely defined and summarized in a set of rules.

The governance model comprises some main pillars. The first one is the governance structure. Since the industrial partner actively runs the Hyperledger network and connects with its clients, a semi-decentralized structure was defined where the industrial partner has more rights than the customers. Furthermore, universal voting rights and a general voting scheme were designed that says that all parties are equal, and decisions are only valid if all parties agree. The industrial partner has the option to promote clients to a higher level that could be useful if one client is also strongly incorporated with the other clients.

The second pillar is the stakeholder definition and identification of their associated roles. In general, there are three possible stakeholders: The industrial partner, the client, and a trust steward that can act on behalf of one of the other parties. In this case, the delegating party needs to define for which occasions the trust steward is authorized in which scope. Furthermore, it is possible to add more actors to the system.

In the next step, the Hyperledger network and channel policy are defined. Procedures for changes in organizations or the infrastructure, such as adding or removing a peer from the system were established.

Last, basic rules for generating, transferring, processing, and interpreting machine and production data were set. These rules provide information on who can write and read the data and contains a set of penalties for breaking the rules.

3.3. Hyperledger network setup

In order to set up a Hyperledger network, some steps need to be undertaken. One of the most fundamental processes are bringing up a network, creation of channels and deployment of chaincode. These steps can be summarized in the following steps (Kam, 2020).

1. Generate the crypto material (https://hyperledger-fabric.readthedocs.io/en/release-2.2/commands/cryptogen.html) for each organization with its own CA, including components (orderer/peer) and users (an admin and a user). Additionally, the consortium genesis block is generated on the system channel
2. Bring up the Certificate Authorities, one for each organization
3. Bring up all the network components based on configuration in docker compose files
4. For each channel:
    a. Generate configuration transaction and anchor peer update transactions for the channel
    b. Generate channel genesis block with ordering service
    c. Join all peers with that channel genesis block
    d. Update anchor peer transactions for each organization on that channel
5. For each chaincode:
    a. Package chaincode
    b. Install chaincode package to each peer
    c. Approve chaincode definition for organizations according to lifecycle endorsement policy requirement
    d. Commit chaincode definition

3.3.1. Peer setup

As already described in the previous Chapter, there are different types of peers, with endorsing, committing, and ordering nodes as well as authorized and unauthorized peers. This Chapter focuses on how to set up a peer that has the roles of an authorized, endorsing and committing peer, and how to connect it with the ledger.

In order to set up a peer and connect it with a channel, the following steps need to be performed:

1. Modify crypto-config.yaml:
   After the crypto material is created the channel or network has to be configured to support this extra peer. Therefore, the “count” variable in the “crypto-config.yaml” will be automatically increased by 1 to indicate the addition of another peer

2. Generating crypto material:
   The “cryptogen extend” command allows extending an existing network, meaning generating all the additional key material needed by the newly added entities. This includes public and private keys. This is a simple process that allows other users to set up a peer quickly

3. The key generation will spawn a new peer and automatically joins it with existing network/channel for further synchronization

   a. Create Docker compose definitions for peers and CouchDB
   b. Create Docker scripts for Organization
   c. After execution: Peer and CouchDB will be resolvable with existing peers

4. Create certificates for the peer:
   To enroll the peer at the certificate authority (intermediate or root) to receive valid X.509 certificates

5. Join a channel:
   Set up environment variables and execute it in CLI

   a. Channel_Name: Identifies the channel
   b. CORE_PEER_LOCALMSPID: Who is the associated Organization
   c. CORE_PEER_TLS_ROOTCERT_FILE: Certificate for permissions
   d. CORE_PEER_MSPCONFIGPATH: Identifies Membership Service Provider
   e. CORE_PEER_ADDRESS: Address of the peer

   After the environment variables are selected, execute the “channel join” command

Typically, these steps are automated in a .sh script that handles each step.

3.3.2. Adding organizations

An own organization needs to be configured on a much higher level than a single peer, however, the steps are very similar. These are the steps to add an organization to a channel:

1. Create a <newOrg>-crypto.yaml file and set the variables for the organization like name or domain
2. Use the cryptogen tool to create crypto material for the organization. It reads from the yaml file from step 1

3. Load the configuration from a configtx.yaml file. This file contains important information, such as:
    a. A CA root cert to establish the organizations' root of trust
    b. A TLS root cert, used by the gossip protocol to identify Org3 for block dissemination and service discovery
    c. The admin user certificate (which will be needed to act as the admin of this organization later on)
4. Bring up new organization's components with docker-compose
5. Prepare the CLI environment using configtxlator-tool. This container has been mounted with the organizations' folder, giving us access to the crypto material and TLS certificates for all organizations and the Orderer Org
6. Fetch the Configuration, that shows the name of the organization, the path of the certificates and key material of each organization, where the certificate authority root file can be found as well as IPs and endpoints for the core peer (see Fig. 7)

```
export CORE_PEER_LOCALMSPID="newOrgMSP"
export
CORE_PEER_TLS_ROOTCERT_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/organizati
ons/peerOrganizations/newOrg.example.com/peers/peer0.org1.example.com/tls/ca.crt
export
CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/peer/organizations/
peerOrganizations/org1.example.com/users/Admin@newOrg.example.com/msp
export CORE_PEER_ADDRESS=peer0.org1.example.com:7051
```

Figure 7: Chaincode snippet: Configuration

7. Issue the command to fetch the latest config block
8. Add the newOrg crypto material
9. Sign and submit the config update
10. Join the channel

These steps are also usually automated using a .sh script. You can find an example of it in the Hyperledger Fabric sample-code repository (https://github.com/hyperledger/fabric-samples/blob/master/test-network/addOrg3/addOrg3.sh).

3.3.3. Channel creation

Channels are created by building a channel creation transaction and submitting the transaction to the ordering service. The channel creation transaction specifies the initial configuration of the channel and is used by the ordering service to write the channel genesis block. While it is possible to build the channel creation transaction file manually, it is easier to use the configtxgen tool. The tool works by reading a configtx.yaml file that defines the configuration of your channel, and then writing the relevant information into the channel creation transaction.

The configtx.yaml file specifies the channel configuration of new channels. The information that is required to build the channel configuration is specified in a readable and editable form in the configtx.yaml file. The configtxgen tool uses the channel profiles defined in the configtx.yaml file to create the channel configuration and write it to the protobuf format that can be read by Fabric.

In particular, the following steps need to be performed:

1. Set the channel variables, such as the channel name
2. Create a channel transaction to set the channel artifacts
3. Create an anchor peer transaction
4. Create a channel
5. Join the channel
6. Update the anchor peers
7. Verify the result

This process is also easily automatable in a .sh script. You can find one in the Hyperledger Fabrics sample-code repository (https://github.com/hyperledger/fabric-samples/blob/master/test-network/scripts/createChannel.sh). It is worth mentioning that we changed these scripts, and organized them in such a way that all the necessary crypto material is generated in one script (setup network), and the starting of the peers, channel creation etc. is done in one script (start-network). This modification allows us to restart the network multiple times, without the need of generating new crypto materials, which in the real world translates into avoiding to change the connection profiles every time the network is reset.

3.3.4. Adding of applications

An application describes any system that interacts with the blockchain. For example, it can be a machine that signs and sends transactions to peers, or it can be a web application that reads data from the blockchain.

In order to connect an application to the system, a connection profile (https://hyperledger-fabric.readthedocs.io/en/release-2.2/developapps/connectionprofile.html) needs to be created. This connection profile contains information about the peers that hold the data from the blockchain as well as information about the chaincode.

In case the application creates and submits transactions, it is necessary to create key material and certificates for the application, as well as enrolling the application (https://hyperledger-fabric.readthedocs.io/en/release-1.3/write_first_app.html#enrolling-the-admin-user) to the channel.

The following steps need to be performed in order to add a new application:

1. The application enrolls the admin user, where the admin credentials are generated from the Certificate Authority.
2. The application enrolls the application user. The application uses the admin user to register and enroll an app user which will be used to interact with the blockchain network.
3. The application prepares a connection to the channel and smart contract which allows the registered user to invoke chaincode functions.
4. The application initializes the ledger by calling “initLedger”-function and to find a set of required peers and submits the transaction to the ordering services.
5. The application invokes each of the chaincode functions to query the ledger.

3.4. Dynamic leasing

Implementing dynamic leasing for machines is an important and innovative use case for the machine manufacturers and their clients. The main benefits are more transparency for the manufacturer and fairer billing for the client.

In the KOSMoS project, we created a reference implementation for this use case. It contains two organizations, the machine manufacturer and one client that runs the machine. The machine manufacturer maintains the Hyperledger network and the specific channel for the client. The machine manufacturer and the customer each will run two peers in order to fulfill the minimal requirements of Byzantine Fault Tolerance (BFT). Connected to this peer is one application to which the machines of the client send production and machine data. The data is being converted and signed in the blockchain connector element on the application. The application receives the data, converts it into a transaction, and signs it. After the transaction is signed, it will be sent to the peers in the channel.

There are in total three different chaincodes initiated that perform operations for the leasing contracts, the tariffs, and the machines and are written in TypeScript. Each chaincode consists of two files - one file for the business logic and one file for blockchain interaction such as the creation of a transaction that contains the data from and for the business logic. Besides, there are regular dependencies and database information.

3.4.1. Machine

The machine chaincode is responsible for any interaction and operation regarding a machine. It comprises logic to create a machine asset and to update its attributes. This machine asset is required in any further interaction, and therefore requires to be created first.

The following figure (Fig. 8) shows the machine.ts file with the contents of a machine asset and its attributes. This allows the user to give it a human-readable name that makes it easier to reference it in the tariff or the production data.

```
import { Crash } from './crash';
import { ProdData } from './prodData';

export interface Machine {
   machineName: string;
   customerId: string;
   machineDesc: string;
   assetId: string;
   balance?: number;
   timestamp: string;
   prodData?: ProdData[];
   crash?: Crash[];
}
```

Figure 8: Chaincode snippet: machine.ts file

Further there is a file for the bill that shows the data for the bill (see Fig. 9).

```
import { Crash } from './crash';
import { ProdData } from './prodData';

export interface Bill {
   assetId: string;
   bill: number;
   processedProdData: ProdData[];
   processedCrashes: Crash[];
}
```

Figure 9: Chaincode snippet: bill file

The crash.ts file contains the information necessary to perform operations for crashes (see Fig. 10)

```
export interface Crash {
   crash: number;
   timestamp: string;
}
```

Figure 10: Chaincode snippet: crash.ts file

The prodData.ts file contains the information necessary to perform operations for the production data (see Fig. 11)

```
export interface ProdData {
   customerId: string;
   machineId: string;
   prodMinutes: number;
   prodPieces: number;
   toolChanges: number;
   travelDistance: number;
   directionChange: number;
   timestamp?: string;
   transactionId?: string;
}
```

Figure 11: Chaincode snippet: prodData.ts file

The following figure (Fig. 12) shows the business logic to create and update a machine from the machine.chaincode.ts file.

```
export class MachineChaincode extends Contract {

   public async initLedger(ctx: Context) {
       console.info('============= START : Initialize Ledger ===========');
       console.info('============= END : Initialize Ledger ===========');
   }

   public async health(ctx: Context) {
       return {status: 'ok'}; // checking for db is done in the contract chaincode
   }

   public async get(ctx: Context, assetId: string): Promise<Machine> {
       const machineAsBytes = await ctx.stub.getState(assetId);
       try {
           return JSON.parse(machineAsBytes.toString());
       } catch (e) {
           if (machineAsBytes.length === 0) {
               throw Errors.httpError(HTTPStatus.NOT_FOUND, `Machine with id ${assetId} not
found.`);
           } else {
               throw Errors.httpError(HTTPStatus.INTERNAL_SERVER_ERROR, `Could not retrieve machine
with id ${assetId}.`);
           }
       }
   }

   public async getAll(ctx: Context): Promise<Machine[]> {
       const query = {
           selector: {
               assetId: {$gt: null},
           },
       };

       const resultsIterator: Iterators.StateQueryIterator = await
ctx.stub.getQueryResult(JSON.stringify(query));
       return await iterateState(resultsIterator);
   }

   public async getByMachineName(ctx: Context, machineName: string): Promise<Machine[]> {
       const query = {
           selector: {
               machineName,
           },
       };

       const resultsIterator: Iterators.StateQueryIterator = await
ctx.stub.getQueryResult(JSON.stringify(query));
       return await iterateState(resultsIterator);
```

```
    }

    public async update(ctx: Context, machineJSON: string): Promise<Machine> {
        let newMachine: Machine = JSON.parse(machineJSON);
        const oldMachine: Machine = await this.get(ctx, newMachine.assetId);

        console.log(newMachine);
        if (oldMachine) {
            if (newMachine.machineName && newMachine.machineName !== oldMachine.machineName) {
                const existingMachine: Machine[] = await this.getByMachineName(ctx,
newMachine.machineName);
                if (existingMachine.length > 0) {
                    throw Errors.httpError(HTTPStatus.BAD_REQUEST, 'Machine with given name already
exists.');
                }
            }

            newMachine = {
                ...oldMachine,
                ...newMachine,
            };

            await ctx.stub.putState(oldMachine.assetId, Buffer.from(JSON.stringify(newMachine)));

        } else {
            throw Errors.httpError(HTTPStatus.NOT_FOUND, 'Machine with given assetId doesn\'t
exist.');
        }

        return newMachine;
    }

    public async create(ctx: Context, machineJSON: string): Promise<Machine> {
        const machine: Machine = JSON.parse(machineJSON);
        const existingMachine: Machine[] = await this.getByMachineName(ctx, machine.machineName);
        console.log('Existing machines:');
        console.log(existingMachine);
        if (existingMachine.length > 0) {
            throw Errors.httpError(HTTPStatus.BAD_REQUEST, `Machine with given name already
exists.`);
        }

        const newMachine: Machine = {
            ...machine,
            prodData: [],
            crash: [],
            balance: 0,
        };
```

```
        await ctx.stub.putState(newMachine.assetId, Buffer.from(JSON.stringify(newMachine)));
        return newMachine;
    }

    public async createProdData(ctx: Context, assetId: string, prodDataJSON: string):
Promise<ProdData[]> {
        const machine: Machine = await this.get(ctx, assetId);
        const prodData: ProdData = JSON.parse(prodDataJSON);

        prodData.transactionId = ctx.stub.getTxID();
        machine.prodData.push(prodData);

        await ctx.stub.putState(machine.assetId, Buffer.from(JSON.stringify(machine)));
        return machine.prodData;
    }

    public async getProdData(ctx: Context, assetId: string): Promise<ProdData[]> {
        const machine: Machine = await this.get(ctx, assetId);
        return machine.prodData;
    }

    public async createCrash(ctx: Context, assetId: string, crashJSON: string): Promise<Crash[]> {
        const machine: Machine = await this.get(ctx, assetId);
        const crash: Crash = JSON.parse(crashJSON);

        machine.crash.push(crash);

        await ctx.stub.putState(machine.assetId, Buffer.from(JSON.stringify(machine)));
        return machine.crash;
    }

    public async getCrash(ctx: Context, assetId: string): Promise<Crash[]> {
        const machine: Machine = await this.get(ctx, assetId);
        return machine.crash;
    }

    public async bill(ctx: Context, billJSON: string): Promise<string> {
        const currentBill: Bill = JSON.parse(billJSON);

        const machine: Machine = await this.get(ctx, currentBill.assetId);

        const processedProdDataSet = new Set(currentBill.processedProdData);
        const processedCrashesSet = new Set(currentBill.processedCrashes);

        machine.prodData = machine.prodData.filter((p: ProdData) => !processedProdDataSet.has(p));
        machine.crash = machine.crash.filter((c: Crash) => !processedCrashesSet.has(c));
```

```
        machine.balance = machine.balance - currentBill.bill;

        return JSON.stringify({status: 'ok'});
    }

}
```

Figure 12: Chaincode snippet: machine.chaincode.ts file

The index.ts file contains logic to publish the asset on the system and contains a simple export function (see Fig. 13).

```
/*
* SPDX-License-Identifier: Apache-2.0
*/

import { MachineChaincode } from './machine.chaincode';

export { MachineChaincode } from './machine.chaincode';

export const contracts: any[] = [MachineChaincode];
```

Figure 13: Chaincode snippet: index.ts file

3.4.2. Tariff

The tariff chaincode manages tariff assets that can be assigned to contracts and machines. Having a distinct tariff asset creates a modular architecture that allows flexibility as it makes it easier to exchange cost factors for a specific machine. The cost factors include, among other things, the cost of production and wear. There are also costs for a machine failure or for maintenance. Some cost factors require additional calculation like the costs per produced pieces for instance. Here, the average cost of produced pieces in relation to the production minutes is calculated and then multiplied with the amount of produced pieces.

The following figure (Fig. 14) shows the tariff.ts file with the contents of a tariff asset. In particular, it contains the attributes of the tariff, such as the basic costs, the costs for production minutes, or costs for expendables and many more.

```
import { CostProducedPieces } from './cost-produced-pieces';
import { MaterialCost } from './material-cost';

export interface Tariff {
   assetId: string;
   tariffName: string;
   basisCost: number;
   costMinute: number;
   costPiece: CostProducedPieces;
   costMovingDistance: number;
   costToolChanges: number;
   costDirectionChanges: number;
   costCrash: number;
   costExpendables: number;
   costMaintenance: number;
   costMaterial: MaterialCost;
   timestamp?: string;
}
```

Figure 14: Chaincode snippet: tariff.ts file

Additionally, the material-cost.ts file shows the data for the material cost (see Fig. 15).

```
export interface MaterialCost {
   aluminum: number;
   titanium: number;
   steel: number;
}
```

Figure 15: Chaincode snippet: material-cost.ts file

Besides the material.cost.ts, the cost-produced-pieces.ts file shows the data required to calculate the cost for produced pieces which are incrementally changing. (see Fig. 16).

```
export interface CostProducedPieces {
   stepSize: number;
   costFactor: number;
}
```

Figure 16: Chaincode snippet: cost-produced-pieces.ts file

The following figure (Fig. 17) shows the business logic to create and update a tariff from the tariff.chaincode.ts file. It also contains logic to find a specific tariff.

```
/*
* SPDX-License-Identifier: Apache-2.0
*/

import { Context, Contract } from 'fabric-contract-api';
import { Iterators } from 'fabric-shim';
import { Tariff } from './assets/tariff';
import { Errors, HTTPStatus } from './utils/errors';
import { iterateState } from './utils/iterator';

export class TariffChaincode extends Contract {

   public async initLedger(ctx: Context) {
       console.info('============= START : Initialize Ledger ===========');
       console.info('============= END : Initialize Ledger ===========');
   }

   public async health(ctx: Context) {
       return JSON.stringify({status: 'ok'}); // checking for db is done in the contract chaincode
   }

   public async get(ctx: Context, assetId: string): Promise<Tariff> {
       const tariffAsBytes = await ctx.stub.getState(assetId);
       try {
           return JSON.parse(tariffAsBytes.toString());
       } catch (e) {
           if (tariffAsBytes.length === 0) {
               throw Errors.httpError(HTTPStatus.NOT_FOUND, `Tariff with assetId ${assetId} not
found.`);
           } else {
               throw Errors.httpError(HTTPStatus.INTERNAL_SERVER_ERROR,
                   `Could not retrieve tariff with assetId ${assetId}.`);
           }
       }
   }

   public async getAll(ctx: Context): Promise<Tariff[]> {
       const query = {
           selector: {
               assetId: {$gt: null},
           },
       };

       const resultsIterator: Iterators.StateQueryIterator = await
ctx.stub.getQueryResult(JSON.stringify(query));
       return await iterateState(resultsIterator);
   }
```

```
    public async getByTariffName(ctx: Context, tariffName: string): Promise<Tariff[]> {
        const query = {
            selector: {
                tariffName,
            },
        };

        const resultsIterator: Iterators.StateQueryIterator = await
ctx.stub.getQueryResult(JSON.stringify(query));
        return await iterateState(resultsIterator);
    }

    public async update(ctx: Context, tariffJson: string): Promise<Tariff> {
        let newTariff: Tariff = JSON.parse(tariffJson);
        const oldTariff: Tariff = await this.get(ctx, newTariff.assetId);

        console.log(oldTariff);
        if (oldTariff) {
            if (newTariff.tariffName && newTariff.tariffName !== oldTariff.tariffName) {
                const existingTariff: Tariff[] = await this.getByTariffName(ctx,
newTariff.tariffName);
                if (existingTariff.length > 0) {
                    throw Errors.httpError(HTTPStatus.BAD_REQUEST, 'Tariff with given name already
exists.');
                }
            }

            newTariff = {
                ...oldTariff,
                ...newTariff,
            };

            await ctx.stub.putState(oldTariff.assetId, Buffer.from(JSON.stringify(newTariff)));
        } else {
            throw Errors.httpError(HTTPStatus.NOT_FOUND, 'Tariff with given assetId doesn\'t
exist.');
        }

        return newTariff;
    }

    public async create(ctx: Context, tariffJSON: string): Promise<Tariff> {
        const tariff: Tariff = JSON.parse(tariffJSON);
        const existingTariff: Tariff[] = await this.getByTariffName(ctx, tariff.tariffName);
        if (existingTariff.length > 0) {
            throw Errors.httpError(HTTPStatus.BAD_REQUEST, `Tariff with given name already exists.`);
        }
```

```
        await ctx.stub.putState(tariff.assetId, Buffer.from(JSON.stringify(tariff)));
        return tariff;
    }
}
```

Figure 17: Chaincode snippet: tariff.chaincode.ts file

The index.ts file contains logic to publish the asset on the system and contains a simple export function (see Fig. 18).

```
/*
* SPDX-License-Identifier: Apache-2.0
*/

import { TariffChaincode } from './tarrif.chaincode';

export { TariffChaincode } from './tarrif.chaincode';

export const contracts: any[] = [TariffChaincode];
```

Figure 18: Chaincode snippet: index.ts file

3.4.3. Contract

The contract chaincode is responsible for the metadata of the contract, such as machine manufacturer, customer, start and end date, and the billing interval. In addition, the parameters for additional services such as maintenance or consumables are specified here. Additionally, it manages the incoming data from the machine and stores it on the blockchain. It receives data of the minutes of production, produced pieces, tool changes, travel distance, direction changes, and crashes. This chaincode also integrates the machineId and tariffId from the two previous chaincodes.

The following figure (Fig. 19) shows the contract.ts file with the contents of a contract asset. In particular, it contains the current status of the contract which depends on the start- and end-dates of the contract as well as attributes of the contract, such as the referenced machine and tariff, the material used, or a versioning.

```
export enum LeaseContractStatus {
   UPCOMING = 'UPCOMING',
   ACTIVE = 'ACTIVE',
   DEACTIVATED = 'DEACTIVATED',
}

export enum LeaseContractBillingInterval {
   WEEKLY = 'WEEKLY',
   MONTHLY = 'MONTHLY',
}

export interface LeaseContract {
   contractName?: string;
   manufacturerId?: string;
   customerId?: string;
   machineId?: string;
   tariffId?: string;
   expendables?: boolean;
   maintenance?: boolean;
   beginDate?: string;
   endDate?: string;
   billingInterval?: LeaseContractBillingInterval;
   nextBillingDate?: string;
   lastBillingOn?: string;

   assetId?: string;
   status?: LeaseContractStatus;
   version?: string;
   timestamp?: string;

   // TODO: the following are not specified in the doc.
   docType?: string;
   material?: string;
}

export const LEASE_CONTRACT = 'lease_contract';
```

Figure 19: Chaincode snippet: contract.ts file

The following figure (Fig. 20) shows the business logic to create and update a contract from the contract.chaincode.ts file. It also contains logic to find a specific tariff.

```
import { Context, Contract } from 'fabric-contract-api';
import { Iterators } from 'fabric-shim';
import { LEASE_CONTRACT, LeaseContract, LeaseContractStatus } from './assets/lease.contract';
import { calculateNextBillingDate, getContractStatus, minDate } from './utils/date.utils';
import { Errors, HTTPStatus } from './utils/errors';
import { iterateState } from './utils/iterator';

export class ContractChaincode extends Contract {

   public async initLedger(ctx: Context) {
       console.info('============= START : Initialize Ledger ===========');

       console.info('============= END : Initialize Ledger ===========');
   }

   public async health(ctx: Context) {
       await this.getAllContracts(ctx); // this checks if the db is working
       return JSON.stringify({status: 'ok'});
   }

   public async create(ctx: Context, leaseContract: string): Promise<LeaseContract> {

       console.info('=========== Create Lease LeaseContract ===========');

       const newLeaseContract: LeaseContract = JSON.parse(leaseContract);

       const existingContracts = (await this.getContractByMachineId(ctx,
newLeaseContract.machineId));

       if (existingContracts) {
           const activeContract =
               existingContracts.find((o: LeaseContract) => new Date(o.endDate) >= new Date());

           if (activeContract) {
               activeContract.endDate = minDate(activeContract.endDate, newLeaseContract.beginDate);
               activeContract.nextBillingDate = minDate(activeContract.endDate,
activeContract.nextBillingDate);
               await ctx.stub.putState(activeContract.assetId,
Buffer.from(JSON.stringify(activeContract)));
           }
       }

       newLeaseContract.nextBillingDate =
minDate(calculateNextBillingDate(newLeaseContract.beginDate,
           newLeaseContract.billingInterval).toISOString(),
           newLeaseContract.endDate);

       newLeaseContract.docType = LEASE_CONTRACT;
```

```
        newLeaseContract.version = '1';
        newLeaseContract.status = getContractStatus(newLeaseContract);

        await ctx.stub.putState(newLeaseContract.assetId,
Buffer.from(JSON.stringify(newLeaseContract)));

        ctx.stub.setEvent('create-lease-contract', Buffer.from(JSON.stringify(newLeaseContract)));

        console.info('============= End Create Lease LeaseContract ===========');

        return newLeaseContract;
    }

    public async getContractByMachineId(ctx: Context, machineId: string): Promise<LeaseContract[]> {
        console.info('=========== Get Lease Contracts By Machine Id ===========');

        const query = {
            selector: {
                machineId,
            },
        };

        const resultsIterator: Iterators.StateQueryIterator = await
ctx.stub.getQueryResult(JSON.stringify(query));

        console.info('=========== End Get Lease Contracts By Machine Id ===========');

        return await iterateState<LeaseContract>(resultsIterator);
    }

    public async get(ctx: Context, assetId: string): Promise<LeaseContract> {
        console.info('=========== Query Lease LeaseContract ===========');

        const contractAsBytes = await ctx.stub.getState(assetId);
        let contract: LeaseContract;

        try {
            contract = JSON.parse(contractAsBytes.toString());
        } catch (e) {
            if (contractAsBytes.length === 0) {
                throw Errors.httpError(HTTPStatus.NOT_FOUND, `Contract with id ${assetId} not
found.`);
            } else {
                throw Errors.httpError(HTTPStatus.INTERNAL_SERVER_ERROR,
                    `Could not retrieve contract with id ${assetId}.`);
            }
        }
```

```
        console.info('========== End Query Lease LeaseContract ==========');

        return contract;
    }

    public async editContract(ctx: Context, leaseContract: string): Promise<LeaseContract> {
        console.info('========== Edit Lease LeaseContract ==========');

        let newLeaseContract: LeaseContract = JSON.parse(leaseContract);
        const oldLeaseContract = await this.get(ctx, newLeaseContract.assetId);

        newLeaseContract = {
            ...oldLeaseContract,
            ...newLeaseContract,
            version: String(Number(oldLeaseContract.version) + 1),
        };

        if (new Date(oldLeaseContract.endDate) <= new Date() && new Date(newLeaseContract.endDate) >=
new Date()) {
            // reactivating deactivated contract
            const beginningOfDay = new Date();
            // To make sure there are no conflicts between the peers
            beginningOfDay.setUTCHours(0, 0, 0, 0);
            const nextBillingDateStartingFromToday =
                calculateNextBillingDate(beginningOfDay.toISOString(),
newLeaseContract.billingInterval).toISOString();

            newLeaseContract.nextBillingDate =
                minDate(newLeaseContract.nextBillingDate, nextBillingDateStartingFromToday);

            newLeaseContract.status = getContractStatus(newLeaseContract);

        } else if (oldLeaseContract.beginDate !== newLeaseContract.beginDate
            && new Date(oldLeaseContract.beginDate) > new Date()) {
            // update to beingDate of upcoming contract
            newLeaseContract.nextBillingDate =
                calculateNextBillingDate(newLeaseContract.beginDate,
newLeaseContract.billingInterval).toISOString();
        }

        await ctx.stub.putState(newLeaseContract.assetId,
Buffer.from(JSON.stringify(newLeaseContract)));

        console.info('========== End Edit Lease LeaseContract ==========');

        return newLeaseContract;
    }
```

```
    public async getAllContracts(ctx: Context) {
        console.info('========== Get All Lease Contracts ==========');

        const query = {
            selector: {
                assetId: {$gt: null},
            },
        };

        const resultsIterator: Iterators.StateQueryIterator = await
ctx.stub.getQueryResult(JSON.stringify(query));

        console.info('========== End Get All Lease Contracts ==========');

        return await iterateState<LeaseContract>(resultsIterator);
    }

    public async getContractsByName(ctx: Context, contractName: string) {
        console.info('========== Get Lease Contracts By Name ==========');

        const query = {
            selector: {
                contractName,
            },
        };

        const resultsIterator: Iterators.StateQueryIterator = await
ctx.stub.getQueryResult(JSON.stringify(query));

        console.info('========== End Get Lease Contracts By Name ==========');

        return await iterateState<LeaseContract>(resultsIterator);
    }

    public async getContractsByNextBillingDate(ctx: Context, nextBillingDate: string):
Promise<LeaseContract[]> {
        console.info('========== Get Lease Contracts By Next Billing Date ==========');

        const query = {
            selector: {
                nextBillingDate: {
                    $lte: nextBillingDate,
                },
                status: LeaseContractStatus.ACTIVE,
            },
        };

        const resultsIterator: Iterators.StateQueryIterator = await
```

```
ctx.stub.getQueryResult(JSON.stringify(query));

        console.info('============ End Get Lease Contracts By Next Billing Date ============');

        return await iterateState<LeaseContract>(resultsIterator);
    }

    public async bill(ctx: Context, assetId: string, lastBillingOn: string, nextBillingDate: string):
Promise<string> {
        console.info('============ Bill Contract ============');

        const contract = await this.get(ctx, assetId);
        contract.lastBillingOn = lastBillingOn;
        contract.nextBillingDate = nextBillingDate;

        if (new Date(contract.endDate) < new Date(lastBillingOn)) {
            // contract has expired, deactivate it
            contract.status = LeaseContractStatus.DEACTIVATED;
        }
        console.info('============ End Bill Contract ============');

        return JSON.stringify({status: 'ok'});
    }

}
```

Figure 20: Chaincode snippet: contract.chaincode.ts file

The index.ts file contains logic to publish the asset on the system and contains a simple export function (see Fig. 21).

```
/*
* SPDX-License-Identifier: Apache-2.0
*/

import { ContractChaincode } from './contract.chaincode';

export { ContractChaincode } from './contract.chaincode';

export const contracts: any[] = [ContractChaincode];
```

Figure 21: Chaincode snippet: index.ts file

3.4.4. Data structure

The following figure (Fig. 22) shows the connection between the different assets that are stored on the ledger. There are five different assets for the contracts, machines, tariffs,

production data and crashes. The separation of the assets allows for a dynamic and modular management of the system.

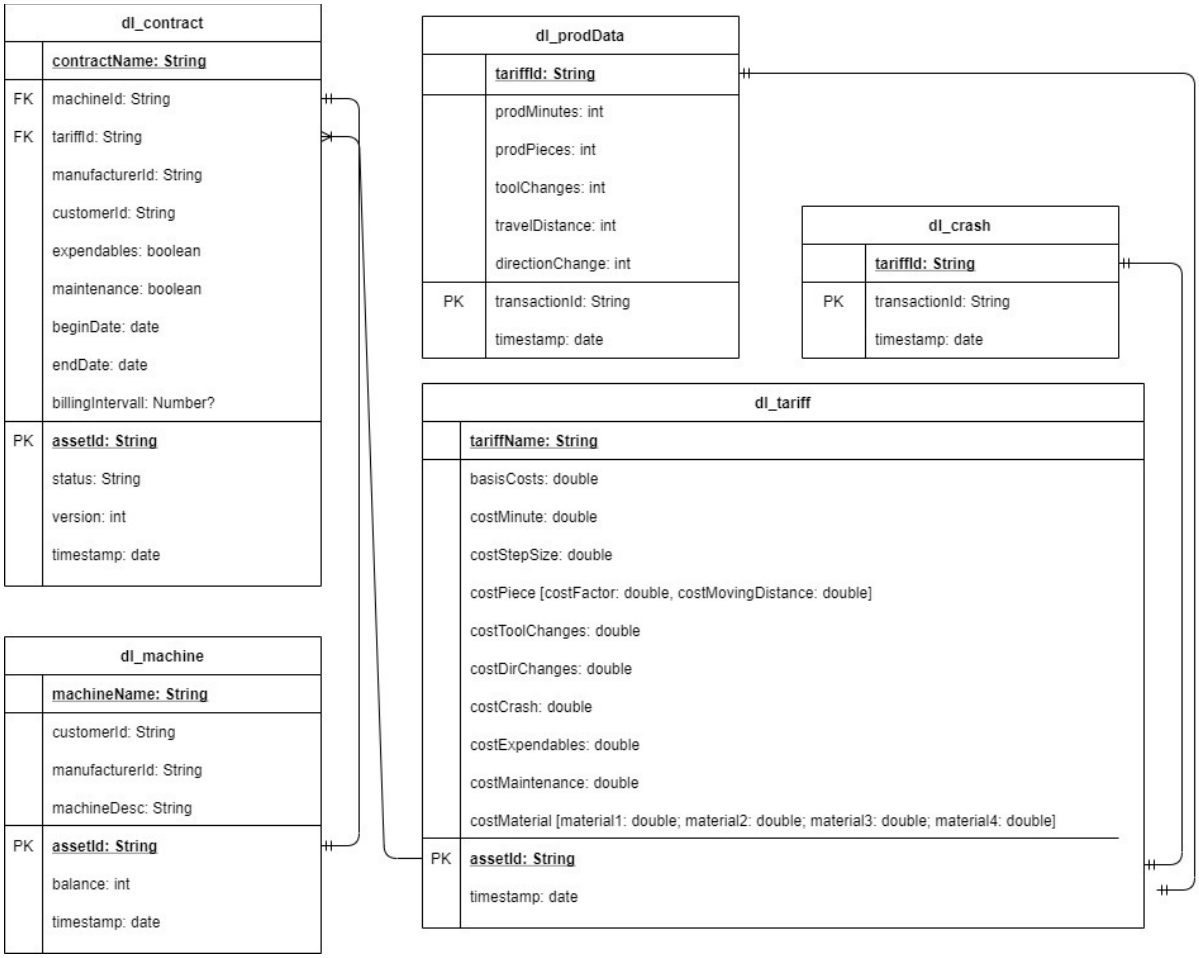

Figure 22: Data structure

3.5. Transparent maintenance

Due to the ongoing nature of the project, this chapter is still under development. The chapter will be published in a final version after the project is finished.

4. Further use cases related to the KOSMoS project

Dynamic leasing (2.2.1) and transparent maintenance (2.2.2) constitute two feasible ways to exploit the potential of DLT in the manufacturing industry. Additionally, there are several other possibilities to improve business processes and increase efficiency using DLT.

To raise awareness for the potential of DLT-based business models in the Industry 4.0 sector, we want to discuss **further use cases** related to the KOSMoS project, divided into **use cases related to Hyperledger Fabric** (4.1) and **Industry 4.0** (4.2) respectively. However, we will not implement these use cases in the context of this project but rather give an overview of further promising applications.

4.1. Use cases related to Hyperledger

This document has already described the advantages of Hyperledger Fabric as a modular, scalable, and secure foundation for offering industrial blockchain solutions in detail. These advantages also have high potential for the following two use cases, **supply chain operations** (4.1.1) and **additive manufacturing** (4.1.2). In both use cases, private transactions and confidential contracts are highly important for businesses, making Hyperledger Fabric the best suited DLT-based solution.

4.1.1. Supply chain operations

Supply chains have grown in complexity along with globalization and customers demanding delivery of products in an inexpensive and timely manner. In addition to that, customers call for more transparency, putting pressure on companies to provide visibility into their supply chains (Kumar et al., 2019). At the same time, companies want to have data sovereignty about sensitive data (e.g., the terms of contract).

This use case aims to offer the possibility to provide a **holistic overview of the history and current location of any product in the supply chain**. This data can then be used to identify inefficiencies within the supply chain as well as increase customer satisfaction and trust through transparency about product provenance. Simultaneously, sensitive data should be securely stored on the blockchain and only be made accessible to trusted participants.

Status quo. In a typical, simplified supply chain, a retailer places an order with a supplier who in turn places an order with a producer. Along this process, problems arise.

First, there are **trust issues** between these parties, e.g., regarding the propriety of production. This is mainly due to a **lack of transparency about the history and quality of the goods**, as companies struggle to provide product provenance.

Second, supply chains are prone to fraud and are exposed to **cybersecurity risks**. If there is unauthorized access to sensitive data, this can potentially cause huge negative economic consequences and distrust in manufacturers.

Stakeholder. The three main stakeholders are the retailer, the supplier, and the producer of the product. However, an outside third party company may be added, aiming to provide advanced analytics on e.g., efficiency, waste, and energy consumption of the entire supply chain (Kumar et al., 2019).

The process starts with the retailer placing an order with the supplier (Fig. 23, step 1). For this purpose, a private channel A is created so only the involved parties can see the exact contract terms. The supplier then places an order with the producer, again via another private channel B (Fig. 23, step 2). This procedure guarantees data sovereignty for all stakeholders, as each member organization maintains full control over who can access different data elements.

Furthermore, a third channel, channel C, is set up involving all parties in order to track the movement of the ordered goods (Fig. 23, step 3). In this channel, every step along the supply chain such as the harvest will be recorded. This opens the possibility for all parties involved to provide transparency about every step of the supply chain, including product information, origin, and authenticity assurance. Additionally, real-time information overview is provided (Fig. 23, step 6), making fast interventions in case of identified complications possible.

In addition to that, smart contracts will enable the automation of supply chain decision making, including the facilitation of real-time order settlement and the subsequent payments. This leads to cost savings and reduced time expenditures for all stakeholders. These advantages are examined in detail in the use cases payments (4.2.1) and automated ordering (4.2.2).

Goals. This use case aims at **providing transparency about the supply chain while simultaneously assuring the data sovereignty of all stakeholders.** The former is achieved by setting up a channel for all affected parties, used for sharing information about the product from the raw resources to the finished product. The latter is achieved by setting up private channels making information only accessible to appropriate users. This setup fulfills the customers' need for transparency while simultaneously eliminating fraud.

Moreover, order settlement and payments should be automated using smart contracts. This leads to a more efficient process, faster delivery times, and finally more satisfied customers.

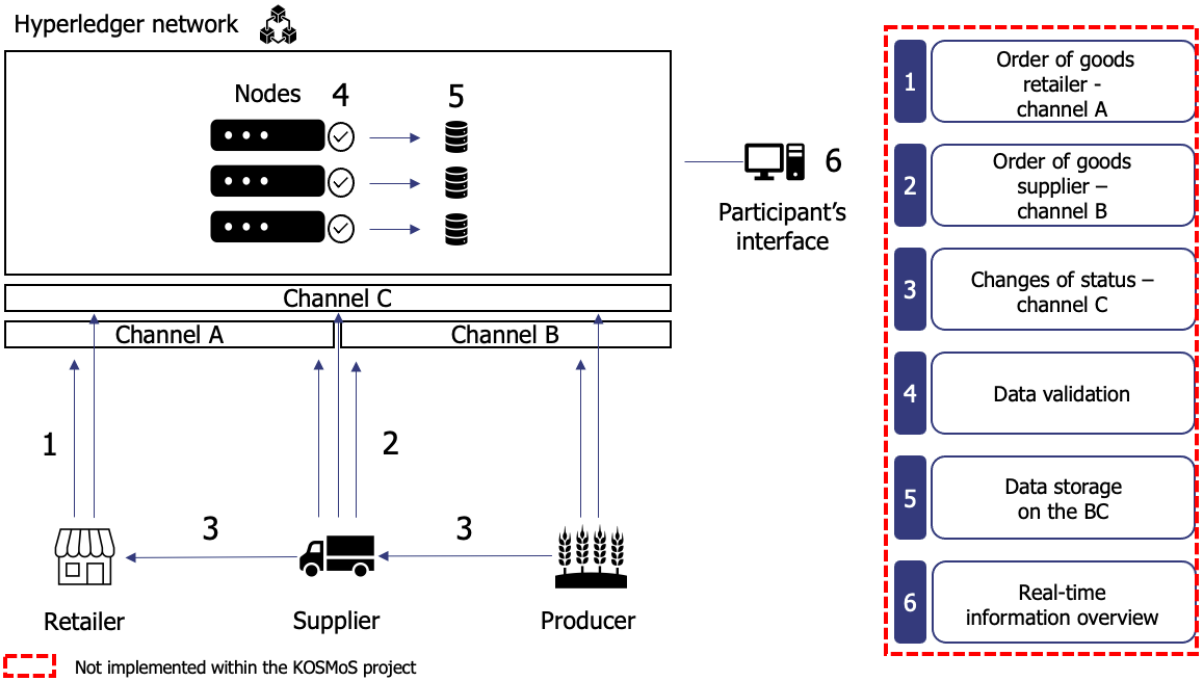

Figure 23: Supply chain operations with Hyperledger Fabric

4.1.2. Additive manufacturing

Additive manufacturing provides a great opportunity to decentralize manufacturing by decoupling development from production (Marx, 2019). However, there are huge concerns over data protection and sovereignty, as the developed print files cause high production, engineering, and development efforts. Therefore, **protecting intellectual property** is key and building trust between the operating stakeholders is very costly. This use case not only aims at overcoming data concerns and reducing implied costs along but also at providing documentation of production data to further smooth operations.

Status quo. The customer sends the CAD (computer-aided design) file, containing process and drafting information for printing, to an engineering service provider generating the final 3D-print file. This file in turn is sent to a 3D-print-partner which then produces the component. During this process, multiple parties are involved assuring the quality and correctness of the process, including site visits and audits to confirm e.g., manufacturing practices, the validity of purchase orders, and others. This implies **huge costs**. Furthermore, there are high **risks of plagiarism**, making the transfer of design data for 3D-printing only economically reasonable if there are appropriate security mechanisms (Holland et al., 2018).

Stakeholders. The three main stakeholders are as mentioned the customer, the engineering service provider, and the 3D-print-partner. The customer still sends the CAD file to the engineering service provider who in turn sends a 3D-printable file to the 3D-print-partner (Fig.

24, step 1). For this process, now private channels are set up. On the one hand, there is channel A to enable private sharing of sensitive information about the transaction between the customer and the engineering service provider (Fig. 24, step 2). On the other hand, another private channel (channel B) is set up to enable the exchange of data on the history of the file (Fig. 24, step 3).

The customer mainly benefits from the data security of his intellectual property, as the blockchain approach implies full data security and sovereignty along the whole production process. At the same time, the 3D-print-partner benefits from the digital product memory of the printer, enabling increased product quality and manufacturing productivity.

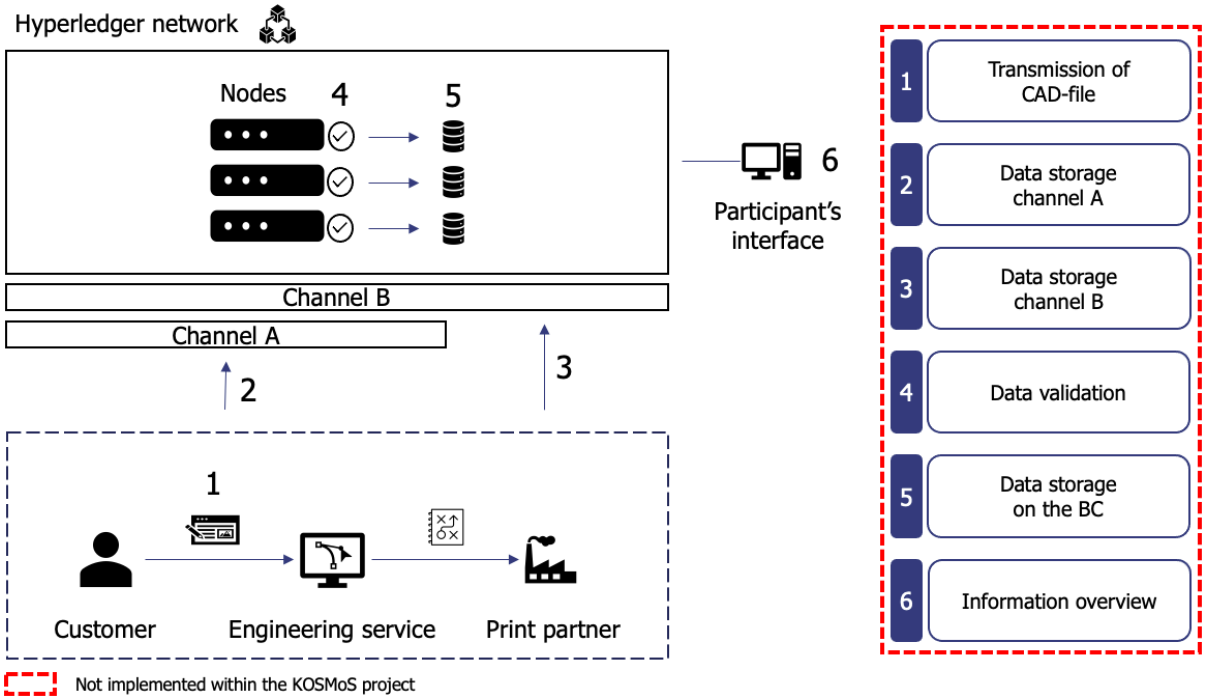

Figure 24: Additive manufacturing with Hyperledger Fabric

Goals. This use case aims at enabling sharing data confidentially by encrypting the 3D-files along the process from design to final production. Along this way, only participants involved in the particular transaction get insight into the complete data set. Furthermore, smart contracts should facilitate the localization of the most appropriate printer (based on different factors like price, location, and availability) and automatically negotiate terms like delivery date and price (Blechschmidt & Stöcker, 2016) which further reduces transaction costs. Finally, as elaborated in the dynamic leasing (2.2.1) and transparent maintenance (2.2.2) use cases, this use case aims at providing production data at every production step. In the case of damaged parts, it can (with the help of the blockchain) be detected, in which step of the process the error occurred.

4.2. Use cases related to Industry 4.0

The following two use cases, **payments** (4.2.1) and **automated ordering** (4.2.2), constitute two further ways to make use of the potential of DLT in the manufacturing industry. Both use cases build upon the use cases intensively discussed in Chapter 2, dynamic leasing (2.2.1) and transparent maintenance (2.2.2). However, in contrast to the other use cases, they are not necessarily based on Hyperledger Fabric but potentially on other blockchains such as Ethereum.

Furthermore, they are closely linked to Industry 4.0 as they are highly dependent on advancements in sensor technology and machine-to-machine communication. The produced data serves as a foundation for smart contracts which in turn facilitate the automation of payments upon successful delivery and the automation of ordering processes.

4.2.1. Payments

To ensure a seamless flow of goods it is crucial to ensure that payments are performed in a fast, inexpensive, and secure way. **However, current payment processes are often very time-consuming, costly, and insecure**.

This use case aims at the automation of payment processes after successful delivery, accelerating payments from several days to fractions of seconds, reducing implied transaction costs, and improving process security. Furthermore, it has the potential to extend the use case dynamic leasing (2.2.1) as it enables the automated real-time payment of leasing fees after usage of a machine.

Status quo. Traditionally, payment transactions were paper-based, requiring manual intervention at every stage. This process is not only time-consuming but also prone to errors and fraud. Larger companies may already have (fully) digitized payment processes, but especially small and medium-sized enterprises often still do not possess such. Moreover, as payment processes of different manufacturers are often disjunct, banks and other financial intermediaries are involved in processing transactions, implying fees and again deceleration of the process. Last, with electronic payment environments growing in recent years also concerns over data security and confidentiality have become a prominent issue. As mentioned in the supply chain operations (4.1.1) use case, unauthorized access to sensitive data possibly causes huge negative economic consequences, and distrust in manufacturers.

Stakeholders. There are two main stakeholders: the buyer and the seller of goods. The process starts with the buyer placing an order of goods. Upon this request, the supplier places

the order to ship the goods. During the shipping, data about the changes in the delivery status are constantly transmitted to the blockchain, enabling tracking along the supply chain (Fig. 25, step 1). Once the goods are delivered, this is confirmed by the recipient, transporter, or sensors (Fig. 25, step 2). If the delivery was successful, the smart contract automatically transfers digital currencies from the buyer's wallet to the seller's wallet (Fig. 25, step 5a). Both wallets are linked to traditional IBAN accounts allowing seamless interaction of fiat currencies and their digital, programmed counterparts (Katilmis et al., 2019). Additionally, a lender could be involved enabling automated financing of the goods. The smart contract would automatically transfer the funds from the lender's wallet (Fig. 25, step 5b). This approach enables very fast and secure processing of transactions. Compared to a couple of days from successful delivery to payment, the payment is now processed within fractions of seconds.

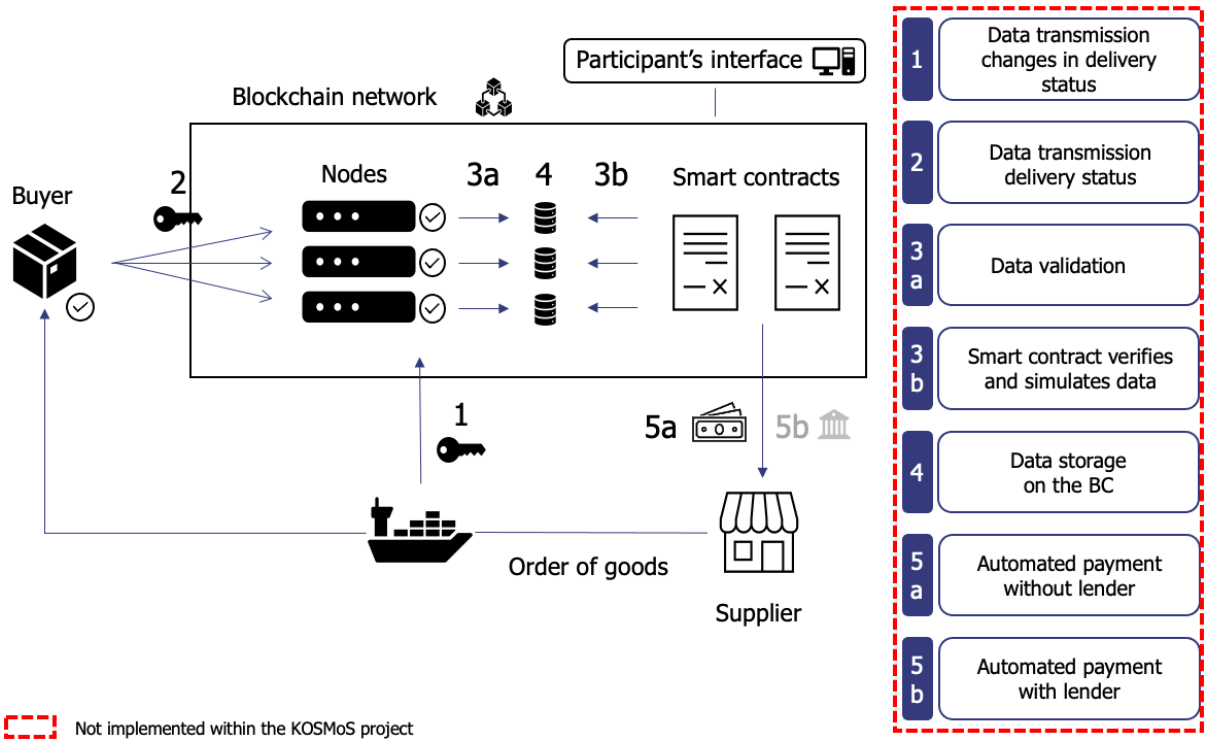

Figure 25: Automated payment on delivery

Goals. This use case aims at **automating transaction processes and thereby reducing transaction costs and time**. This is achieved by smart contracts triggering a payment between the wallets of the buyer and the seller upon delivery of the ordered goods. As stated above, these wallets are matched to traditional IBAN accounts to ensure a seamless transfer of money. Furthermore, this use case could also provide a solution for the pay-per-use business model as elaborated in the use case of dynamic leasing (2.2.1). It has the potential for automated real-time payment of the owner of the machine after usage.

4.2.2. Automated ordering

The use case smart maintenance (2.2.2) has the potential to perform maintenance very targeted and efficiently based on a well-documented maintenance history. However, this cannot prevent parts of the machine from breaking, requiring a replacement. At this point, automated ordering comes into play.
This use case aims at further smoothing manufacturing processes by making use of machine sensors detecting parts that are about to default. Upon such events, **spare parts are ordered automatically, implying a fast and inexpensive way to ensure the seamless production of goods**.

Status quo. Replacements of defective parts of a machine initially require the detection of a defect. Most machines are equipped with lots of sensors making the detection of defective parts possible in advance of a failure event. However, due to insufficient data evaluation often a defect is only recognized after a machine shutdown. The associated costs of a production shutdown are often very high in the manufacturing industry. After the detection of a defective part, the respective spare part has to be ordered. Currently, this implies high administrative efforts (including manual ordering and finding the right supplier) and associated time delays.

Stakeholders. The main stakeholders involved are the machine owner, the industrial partner, and the supplier of the spare parts. Additionally, as in the smart maintenance use case, there can be a third party company performing the actual maintenance. The machine's sensors regularly transmit data to the blockchain (Fig. 26, step 1). If it is detected that a part is about to default a smart contract is triggered. The smart contract then automatically places an order with a supplier of the respective spare part (Fig. 26, step 4). The spare part is then delivered to the machine owner (Fig. 26, step 5). Ideally, the affected spare part is replaced before the machine shuts down (Fig. 26, step 6). This process has various advantages for the stakeholders. First, as the spare part is ordered automatically, the machine owners' administrative costs are reduced compared to a manual order. Second, as the spare part is replaced very fast (ideally ahead of the failure event), machine downtimes are minimized (Babich & Hillary, 2020). This also implies a minimization of the associated costs of a production shutdown for the machine owner. Third, the minimization of machine downtimes also diminishes cases of unplanned maintenance which would have to be conducted by the industrial partner or the third party maintenance company. Last, special rules can be predefined for a smart contract, including e.g., ordering the cheapest spare part, further decreasing costs for the machine owner.

Goals. The aim of this use case is to further smooth the manufacturing process by automatically ordering a spare part upon a detected event of an upcoming default. This

approach minimizes production shutdowns and associated costs as the spare parts are ordered ahead of the failure event. In addition, the automation of the ordering process reduces time and cost expenditures related to the manual ordering and selection of an appropriate supplier and diminishes cases of unplanned maintenance. Finally, the process can be further smoothed by automating the payment of the supplier upon successful delivery of the spare part (described in 4.2.1), again decreasing administrative expenses.

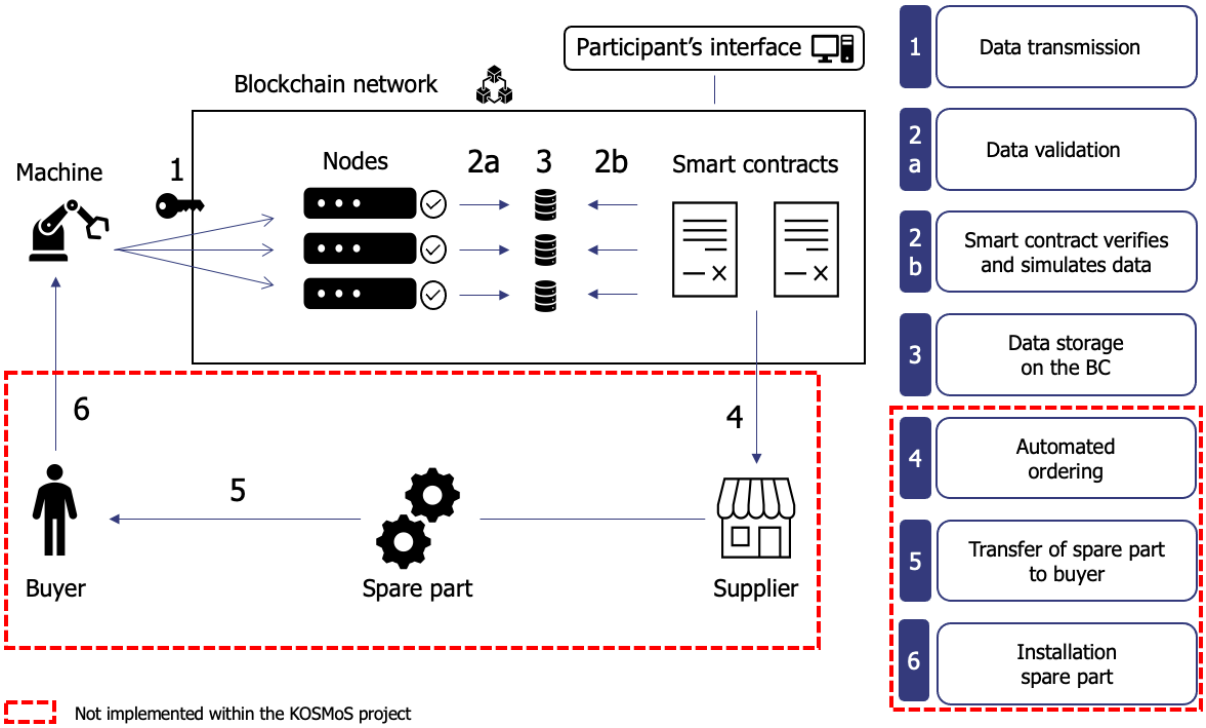

Figure 26: Automated ordering

4.2.3. Others

The following two use cases add on the above-described use cases supply chain operations (4.1.1) and automated payment on delivery (4.2.1). While the use case **automated payment on delivery with collateral** (4.2.3.1) extends the concept of automated payments, the use case **Track and Trust by Datarella** (4.2.3.2) gives a real-world implementation of the advantages discussed in the supply chain use case.

Automated payment on delivery (with collateral). The use case of payment automation (4.2.2) enables a fast, inexpensive, and secure way of payment processes to ensure the seamless transactions of goods. But what happens if one of the involved parties fails to fulfill its contractual obligations, referred to as delivery risk?

This use case aims at **mitigating delivery risks for all involved parties**, by implementing a DLT solution based on a smart contract acting as an escrow.

Status quo. Delivery risk refers to events, where one party involved does not fulfill his contractual obligations. In a typical transaction in the manufacturing industry, this would mean either the asset is not delivered (seller side) or the asset is not paid (buyer side). A possible solution to that would be a third party escrow. However, this approach first implies high fees and second, there are issues regarding the trustworthiness of traditional escrows.

Stakeholders. The main stakeholders are identical to the automation of payments without collateral (4.2.1), namely the buyer and the seller of a good. In this use case, however, the stakeholders benefit in a completely different way. In a simplified process, the product transaction starts with the seller setting up the programmed terms of the smart contract (Fig. 27, step 1). The buyer agrees upon the predefined terms, and the contract is concluded. If the buyer now wants to place an order with the seller, the buyer is required to first deposit collateral (i.e., at least the monetary equivalent of the product price) within the smart contract (Fig. 27, step 2). Here the smart contract acts as an escrow (Hasan & Salah, 2018). After the successful deposit of the collateral, the seller receives a delivery request (Fig. 27, step 3). The product is then shipped to the buyer. The buyer confirms the successful delivery (Fig. 27, step 4), the seller is paid automatically with the deposited collateral (Fig. 27, step 5), and the product is handed over to the buyer (Fig. 27, step 6).

This process has one main advantage for both parties: the reduction of delivery risks. The buyer benefits from the mitigated risk of the seller not delivering the ordered goods upon advance payment. If the seller fails to do so, the buyer will be refunded his collateral. The seller benefits from the mitigated risk of not receiving payment after successful delivery. To receive the ordered goods, the buyer has to confirm the successful delivery which automatically triggers the settlement of the payment with the deposited collateral.

There are two specific advantages of an “escrow smart contract” over a traditional third party trustee. On the one hand, there are no transaction fees and on the other hand, trust-issues are diminished, as all parties agree on the predefined rules of the smart contract.

Goals. This use case extends the possibilities DLT solutions offer for automating payment as described in 4.2.2. It aims at mitigating the delivery risk of transactions in the manufacturing industry without involving a third party trustee. This is achieved by introducing a **smart contract providing escrow services**. It automates the payment to the seller from previously deposited collateral of the buyer upon the confirmation of successful delivery through the buyer. This approach is very secure and cost-efficient.

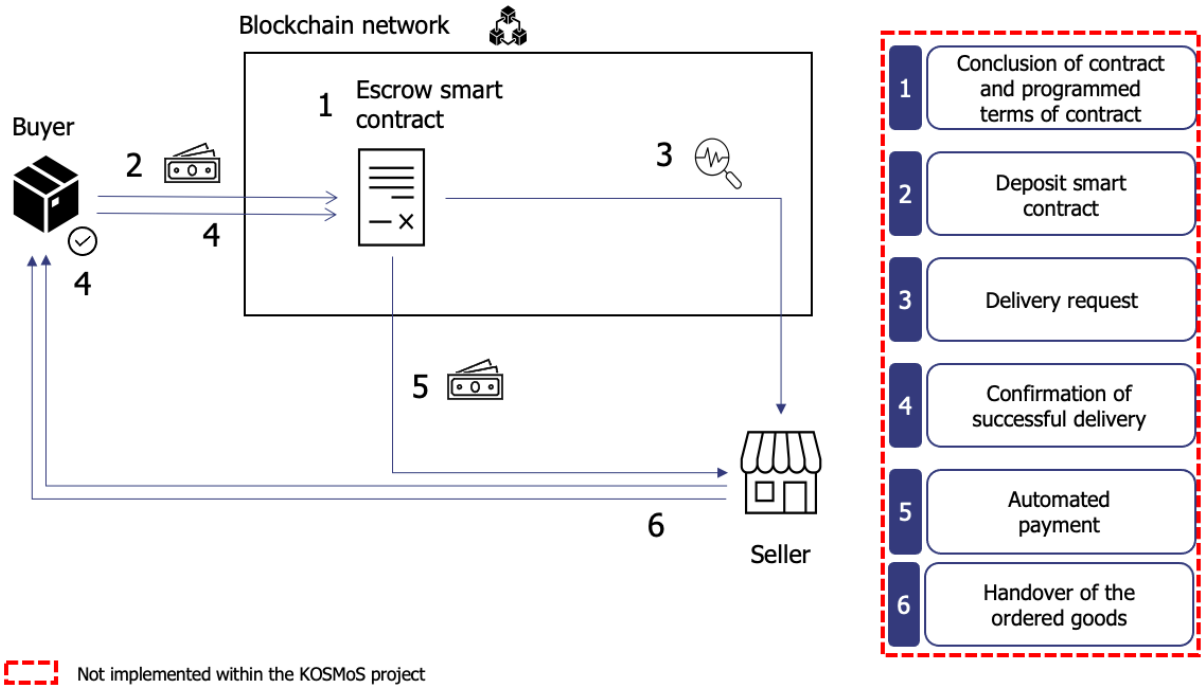

Figure 27: Automated payment on delivery (with collateral)

Track & Trust by Datarella. The use case supply chain operations (4.1.1) described how DLT can be used in supply chains to increase transparency and establish trust between the parties involved. This use case builds on these advantages within the specific area of humanitarian aid. Track and Trust is a humanitarian supply chain for aid supplies in disaster zones developed by the German blockchain expert Datarella. It aims at solving trust issues between the multiple parties involved and providing full transparency over the supply chain, leading to a more efficient process and better planning opportunities for future crises (for further information see Datarella, 2020).

Status quo. All over the globe, there are people requiring humanitarian assistance. In order to provide them with life-saving supplies (like food, water, medicine, or shelter), humanitarian supply chains were set up. However, they are facing extremely challenging circumstances, including high time pressure and operating in the most demanding environments (like warzones and areas after natural disasters). Furthermore, there are difficulties in forecasting demand, as first, every crisis is unique, and second, there is a lack of historical data. The latter is mainly due to individual and manual documentation of tracking of goods and shipments (if done at all). This process is very prone to human error and fraud (Kothe, 2020). Moreover, there are various multinational parties involved, who do not necessarily trust each other (partly again because of the untransparent process). These trust issues complicate cooperation and call for a single source of truth.

Stakeholders. As mentioned, there are various multinational stakeholders involved, including governmental and non-governmental aid organizations. There are usually five major classes of stakeholders (Esa, 2019). First, there is the humanitarian officer, initiating the aid shipment. Second, a supplier obtains and organizes goods in shipments. Third, these goods are handed over to the logistics partner, who is responsible for transporting goods until handing off to the consignee at the port of entry. The consignee then contracts with the implementing partner for last-mile transport. The implementing partner finally arranges the delivery of aid goods to people in need. However, this process implies the mentioned trust issues between the parties involved as well as missing or faulty documentation of the tracking of goods and shipments. Here Track and Trust comes into play. Track and Trust is a supply chain solution based on a private blockchain that stores every step of the process. First, it includes events that define how many of which goods to be delivered, in which area, and due to which disaster (including contractors). Second, an agreement between the humanitarian aid organization and the subcontractors to seal cooperation and responsibilities in the supply chain is added. Last, it includes handovers to document the transfer of responsibility including shipment details. This approach ensures full transparency of how many goods are or were being shipped to a disaster area. The stakeholders now have a better overview while supplying and better input for future planning. Furthermore, the blockchain serves as a single source of truth, establishing trust between the parties involved.

Goals. This use case aims at solving existing problems in humanitarian supply chains namely trust issues between the various multinational parties involved as well as missing or faulty documentation of the tracking of goods and shipping. Track and Trust establishes trust through its tamperproof blockchain solution and enables transparency about single steps within the supply chain. This leads to a more **efficient and cost-effective distribution of supplies** and finally to more saved lives in areas where aids are essential for surviving.

5. Additional information about the KOSMoS project

You can find further details about the KOSMoS project on the following page [GERMAN]: https://www.kosmos-bmbf.de/.

The consortium consists of the following partners:

PROJECT PARTNER		FOCUS
	**ASYS Automatisierungssysteme GmbH** *Dornstadt*	Maintenance concepts; product-accompanying proof of quality
	**Datarella GmbH** *Munich*	Development of blockchain and smart contracts
	**Frankfurt School Blockchain Center** *Frankfurt*	Blockchain development; cross-company business models
	**Institut für Cloud Computing und IT-Sicherheit** *Furtwangen*	Data protection / security in the acquisition of production data
	**inovex GmbH** *Karlsruhe*	Analysis services for collected data; predictive maintenance solution
	**Institut für Steuerungstechnik der Werkzeugmaschinen und**	Process modelling; linking digital and physical components

	Fertigungseinrichtungen (ISW) *Stuttgart*	
ONDICS Apps und Webs für Ihre Prozesse	**Ondics GmbH** *Esslingen*	Edge-solutions for the communication between shopfloor and blockchain
Schütte	**Alfred H. Schütte GmbH & CO. KG** *Cologne*	Cross-company, transparent maintenance concepts
SW Technology People	**Schwäbische Werkzeugmaschinen GmbH** *Schramberg*	Audit-proof billing models through dynamic leasing

6. References

Babich, V., & Hilary, G. (2020). OM Forum - Distributed Ledgers and Operations: What Operations Management Researchers Should Know About Blockchain Technology. Manufacturing & Service Operations Management, 22(2), 223-240.

Blechschmidt, B. & Stöcker, C. (2016). *How Blockchain Can Slash the Manufacturing "Trust Tax"*. https://manufacturing.report/Resources/Whitepapers/b9492551-a132-4451-8eda-182ebd3a2780_Whitepaper-manufacturing.pdf

Datarella. (2020). Track & Trust. DATARELLA. https://datarella.com/track-trust/

Esa. (2019). *Track & Trust - Track & Trust: Space Linked Tracking*. https://business.esa.int/projects/track-trust

Glaser, F., Hawlitschek, F., & Notheisen, B. (2019). Blockchain as a Platform, in: Treiblmaier, H., Beck, R. (Eds.), Business Transformation through Blockchain, vol. 29. Springer International Publishing, Cham, pp. 121–143.

Gross, J., Lichti, C., Schäffner, M., & Sandner, P. (2020a). *Which Blockchain to Choose in a Consortium?* https://medium.com/@jonas.ku1994/which-blockchain-to-choose-in-a-consortium-604b0126c16e.

Gross, J., Lichti, C., Schäffner, M., & Sandner, P. (2020b). *Consensus Mechanisms in Consortia Blockchains.* https://constantin-lichti.medium.com/consensus-mechanisms-in-consortia-blockchains-f22bfc03fc63

Hasan, H.R., & Salah, K. (2018). Blockchain-Based Solution for Proof of Delivery of Physical Assets. In: Chen S., Wang H., Zhang LJ. (eds) Blockchain – ICBC 2018. ICBC 2018. Lecture Notes in Computer Science, vol 10974. Springer, Cham. https://doi.org/10.1007/978-3-319-94478-4_10

Höfelmann, D., & Sandner, P. (2019). *How Should Companies Select a Specific Blockchain Framework? FSBC Working Paper.* https://medium.com/@philippsandner/how-should-companies-select-a-specific-blockchain-framework-8d07eb7875a6.

Holland, M., Nigischer, C., & Stjepandić, J. (2018). Intellectual property protection and licensing of 3d print with blockchain technology. In Transdisciplinary Engineering Methods for Social Innovation of Industry 4.0 (p. 103). The Netherlands: IOS Press.

Kam, K.C. (2020). *Add a Peer to an Organization in Test Network (Hyperledger Fabric v2.2).* https://kctheservant.medium.com/add-a-peer-to-an-organization-in-test-network-hyperledger-fabric-v2-2-4a08cb901c98

Katilmis, A., Forster, M., Chen, H., & Cheng, Y. (2019). *CashOnLedger: The key to using blockchain will be getting the Euro in Smart Contracts.* https://medium.com/@cashonledgertech/cashonledger-the-key-to-using-blockchain-will-be-getting-the-euro-in-smart-contracts-cd40dad638b6

Kothe, P. (2020). *Track & Trust – One Of The Best Supply Chain Solutions at Block.IS.* https://datarella.com/track-trust-one-of-the-best-supply-chain-solutions-at-block-is/

Kumar, A., Liu, R., & Shan, Z. (2019). Is Blockchain a Silver Bullet for Supply Chain Management? Technical Challenges and Research Opportunities. *Decision Sciences, 51*(1), 8–37. https://doi.org/10.1111/deci.12396

Marx, K. (2019). *Lord of the data: How the Blockchain ensures smart production processes*. https://engineered.thyssenkrupp.com/en/lord-of-the-data-how-the-blockchain-ensures-smart-production-processes/