Institute of Architecture of Application Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Conceptualizing and Implementing a Transactional Model for Cross-Chain Smart Contract Invocations

Patrick Dabbert

Course of Study:	Informatik
Examiner:	Prof. Dr. Dr. h. c. Frank Leymann
Supervisor:	Ghareeb Falazi, M.Sc.
Commenced:	October 1, 2021
Completed:	April 1, 2022

Abstract

The trustless nature of smart contracts and blockchains make them interesting for use between organizations, where the participants do not want to blindly trust each other. Often such processes between organizations need to be implemented as business transactions. The rapid development in blockchain technology has lead to a big variety of different blockchains, with different properties and feature sets. If two businesses that use different blockchains want to interact with each other, this leads to business transactions, involving smart contract invocation over different blockchains. The transaction semantics of such cross chain smart contract invocations are unclear. In this thesis we present an adjusted saga pattern as transaction model for cross chain smart contract invocations. We further present a BPMN extension that allows us to utilize the capabilities of existing process engines for the saga pattern, and allows for integration of cross chain smart contract invocations in existing processes.

Kurzfassung

Smart Contracts und Blockchains verlangen nur eingeschränktes Vertrauen von Seiten des Nutzers. Das macht ihren Einsatz in Prozessen zwischen Organisationen zu einer interessanten Möglichkeit, wenn sich die Beteiligten nicht gegenseitig blind vertrauen wollen. Solche Prozesse zwischen Organisationen müssen oft als Business Transaktionen implementiert werden. Die rasante Entwicklung im Feld der Blockchain Technologien hat zu einer großen Menge verschiedener Blockchains mit verschiedenen Eigenschaften und Funktionen geführt. Wollen zwei Organisationen, die unterschiedliche Blockchains nutzen, miteinander interagieren, kann dies zu Business Transaktionen führen, die Smart Contracts auf diesen verschiedenen Blockchains enthalten. Die Transaktionseigenschaften solcher Cross Chain Smart Contract Invocations sind nicht ausreichend untersucht. In dieser Arbeit präsentieren wir ein angepasstes Saga Pattern als Transaktionsmodel für Cross Chain Smart Contract Invocations. Außerdem präsentieren wir eine BPMN Erweiterung, die es ermöglicht, existierende Process Engines zur Implementierung des Saga Pattern einzusetzen. Zusätzlich ermöglicht dieses Vorgehen die Integration von Cross Chain Smart Contract Invocations in existierende Prozesse.

Contents

List of Figures

List of Tables

1 Introduction

Blockchains are a new but successful technology. This has lead to the creation of different blockchains with different properties and features. One of those features is the possibility of defining and executing smart contracts. The trustless nature of blockchains makes smart contracts interesting for integration into business processes. The transaction properties of the execution of smart contracts on a single blockchain are clear. A look into the blockchain's specification is usually all that is needed. The same can not be said for the invocation and interaction of multiple smart contracts on multiple blockchains.

In this thesis we discuss the transactional properties of cross chain smart contract invocations and propose the saga pattern as a possible transactional model for cross chain smart contract invocation.

In Chapter 2 we introduce the necessary concepts to understand this thesis. We introduce blockchains and smart contracts and the idea of blockchain interoperability. We also introduce traditional transaction processing and business transactions and their properties.

In Chapter 3 we look into why one would want to use blockchains and smart contracts for business processes by looking into an example scenario. We then take a look at the transaction semantics of smart contracts on a single blockchain and show how this is different when multiple blockchains are involved. We also look at existing work and why it isn't enough to solve our problem.

In Chapter 4 we then propose a modified saga pattern as solution for the problem. We also introduce a BPMN extension to leverage the capabilities of existing process engines for this and take a look at the SCIP protocol by Falazi et al., that we suggest as a way to communicate between the process engine and the blockchains.

In Chapter 5 we take a look at a proof of concept implementation and the involved parts.

In Chapter 6 we look at the resulting model, it's advantages and disadvantages, and point out directions for further research.

2 Background

In this Chapter we will introduce the necessary concepts to understand this thesis. First we will introduce blockchains and smart contracts, then we will introduce business transactions.

2.1 Blockchain

In this Section we will introduce blockchains and smart contracts. First we take a look at blockchains in general, and will then discern between permissioned and permissionless blockchains. Afterwards we will look at smart contracts.

2.1.1 What is a Blockchain?

Bitcoin [Nak08] was the first Blockchain. It was invented as a way of transferring electronic cash. For this it starts with a distribution of money between accounts and logs all money transactions between accounts. The innovation in this was, that all those transactions (which from now on we will call blockchain transactions to distinguish them from business transaction discussed in Section 2.2) are stored in a distributed ledger, without any central authority. Instead, a decentralized consensus algorithm is used. In the case of Bitcoin the consensus algorithm is called *proof of work*.

The blockchain transactions are stored in so called blocks. Each block contains multiple blockchain transactions, and a hash of the previous block. The chain in the term blockchain comes from those hashes "linking" the blocks together. See Figure 2.1 for an illustration. With the consensus mechanism a node is selected (in case of proof of work, the first one to solve a computationally heavy task) which decides which blockchain transactions are added to the next block. For the work done, the node gets paid a reward, and usually takes a fee from the account sending money, for including a blockchain transaction.

Since there are different blockchains with different technical details, it is useful to describe the term blockchain in a broader way.

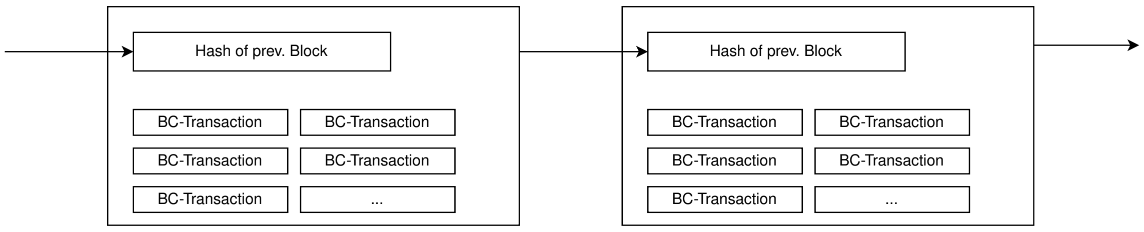

Figure 2.1: Simplified illustration of the structure of a blockchain. Based on a similar figure from [Nak08]. Each block contains the hash of the previous block in the chain and a set of blockchain transactions.

Tai et al. list three ways to view blockchains [TEK17] :

1. "A blockchain is a peer-to-peer protocol for trustless execution and recording of transactions secured by asymmetric cryptography in a consistent and immutable chain of blocks – the blockchain developers and technology view."
2. "A blockchain is a shared append-only distributed database with full replication and a cryptographic transaction permissioning model – the IT architect and data management view."
3. "A blockchain is a shared decentralized ledger, enabling business disintermediation and trustless interactions, thereby lowering transaction costs – the business executive and applications view."

We look at blockchains from the IT architect's perspective. So we see it as a distributed append only database with a cryptographic permissioning model.

2.1.2 Permissioned and Permissionless Blockchains

There are two types of blockchains. Permissionless, also called public, and permissioned ones, also called private.

Nist[YMRS19] describes permissionless blockchains as blockchains, in which everyone can participate. There is no central authority that decides who is allowed to read or write to the blockchain.

Permissioned blockchains are described as blockchains, in which an authority decides who can publish blocks or issue transactions. In those blockchains it can be possible to limit read and write access to the chain. This means it is possible to use them in scenarios with confidential data. This makes them a good tool for cooperation between organizations, since they allow for tighter control than permissionless blockchains, but still don't need as much trust between the organizations, as with traditional methods. The fact, that not everyone can write to the chain, allows for more time and energy efficient consensus mechanisms, which is another advantage. A big disadvantage of permissioned blockchains is, that the participants need to trust the deciding authority. If participants have to be authorized, this also often means a loss of anonymity.

2.1.3 What is a Smart Contract?

Some Blockchains, for example Ethereum [But+14] and Hyperledger Fabric [ABB+18] allow for smart contracts.

A smart contract on a blockchain is code, that is stored on the blockchain and on invocation is executed by the network peers. It is immutable after it is deployed and works exactly as specified [WOY+19].

The Idea behind smart contracts is older than blockchain technology and was already proposed in 1996 by Szabo[Sza96]. Szabo describes the idea of using computers, algorithms and cryptography to ascertain the observance of contracts.

According to Szabo a contract should be designed as *observable* and *verifiable*, meaning it should be possible to observe and proof that a contract was performed or breached by the participants, *enforceable*, ideally self enforcing to minimize the need for enforcing the contract, and *privy*, meaning the involvement of third parties should be minimized as much as possible, both in the way of knowing about the contents and control over how the contract is performed [Sza18].

In a way one can already call the way money is sent over blockchains like Bitcoin an implementation of a smart contract [WOY+19]. Bitcoin even allows for limited scripting ability[But+14], but is not Turing complete. In this thesis we want to focus on Turing complete implementations of smart contracts.

In permissionless blockchains privity isn't given, because of the distributed and open nature of those blockchains, and the fact that all information stored in the blockchain is accessible by all participants. In a way this is a necessity, to allow them to actually execute and verify the contracts.

Other than privity, blockchains are a great way to realize such smart contracts. And with permissioned blockchains, even privity can be achieved, when really needed.

2.1.4 Blockchain Interoperability and Cross Chain Smart Contract Invocation

Traditionally blockchains are self-contained systems. Two different blockchains, like Bitcoin and Ethereum, are unable to directly communicate with each other. This means that if you choose a blockchain to work with, there is no easy way to later move to another blockchain. Communication between blockchains could help with switching chains. Blockchain interoperability could also help to scale applications, by utilizing multiple chains for problem-solving, or help different organizations, that decided on different blockchains, when cooperating.

Schulte et al.[SSFB19] discerns two levels of blockchain interoperability: cross-blockchain token transfer and cross-blockchain smart contract invocation.

Cross chain token transfer is the ability to transfer tokens, for example cryptocurrency, from one blockchain to another. This can be in the form of an exchange between the cryptocurrency of blockchain A and blockchain B, where two participants exchange their currency in a decentralized manner. Another kind of token transfer is one, where no exchange happens, but the data of the token on blockchain A is replicated on blockchain B and then the token on blockchain A is destroyed. Token transfers have to be atomic, meaning they are executed completely or not at all, to prevent duplicates. They also should happen in a trustless manner. [SSFB19]

Cross chain smart contract invocation is the ability to invoke smart contracts across different blockchains. This would make it possible to create applications spanning multiple blockchains. [SSFB19]

These cross chain smart contract invocations could take different forms. They could for example consist of multiple smart contracts, that are being invoked by a middleware. They could also be using the same middleware to communicate with each other. Another approach could be using new blockchains, that implement mechanisms to directly call smart contracts on other blockchains.

2.2 Transactions and Transaction Processing

In this section we will look at transactions and transactional models.

2.2.1 ACID

A business transaction is a process between people or enterprises. For example buying something or booking a hotel is a transaction.[BN09]

In contrast blockchain transactions are more like a request to the blockchain to execute a transaction. This corresponds to what Bernstein and Newcomer[BN09] call a *request message*.

Often business transactions are aided by computers, for example online shopping. This is called transaction processing. In such a case, we usually have multiple programs that concurrently use shared distributed and replicated resources (e.g. a database).[BN09]

From this multiple requirements arise. For example, when booking rooms in a hotel, for every time slot, there should only be one reservation per room.

Parts of those processes, or even the full process are done using computer programs, so called transaction programs. Those transaction programs can be part of a distributed system as well. [BN09]

This means that those transaction programs, which we from now on simply call transactions, have to fulfil certain requirements.

One of the first concretizations of such requirements to transactions is the ACID transactional model [GR92].

The ACID transactional model postulates 4 requirements for a transaction:

Atomicity, Consistency, Isolation and Durability.

Atomicity: A transaction is run either fully or not at all. There are no intermediate states. It either is run completely and commits, or it aborts and has no effect on the system.

Consistency: The transaction moves the system from one consistent state to another.

Isolation: The internal state of a transaction is isolated, and only exists for the transaction. This means that two concurrently running, uncommitted transactions won't affect each other.

Durability: If a transaction commits, its effects stay in the system, and can only be changed by another transaction.

To ensure those requirements, different protocols and techniques like resource locking are used.

One special case of transactions is a transaction that has steps that are themselves transactions. Such a transaction is called *nested transaction*.

2.2.2 Saga Pattern

One special kind of transaction are Long lived transactions (LLTs). LLTs are transactions that take a relative long time, and if they get implemented as an ACID transaction, this can lead to long time locking of resources. But if resources are locked for a long time, this can lead to performance problems, since other processes might need access to the locked resources. [GS87]

Garcia-Molina and Salem[GS87] propose sagas as a solution for the problem of LLTs.

Formally a saga is described as a sequence of ACID transactions:

$$T_1, T_2, ..., T_n$$

and corresponding compensating ACID transactions

$$C_1, C_2, ..., C_{n-1}$$

For which is guaranteed that either the sequence

$$T_1, T_2, ..., T_n$$

or the sequence

$$T_1, T_2, ..., T_j, C_j, C_{j-1}, ..., C_1 \quad (0 \leq j < n)$$

is executed.

The compensating transaction C_i undoes the work done by transaction T_i. [GS87]

A saga basically is a transaction, that is broken up into several smaller sub transactions. Instead of fulfilling ACID for the overall transaction, each sub transaction has an associated compensation transaction. The sub transactions get executed, and if one fails, each successfully executed transactions effect is compensated with its compensation transaction.

A saga is a special form of nested transaction, with only two levels of nesting. Its outer level does not fulfil atomicity, or isolation. [GS87]

Sagas solve the problem of LLTs, by only locking resources for the sub transactions.

While a saga does not fulfil ACID, it is often enough for business transactions[BN09]. For example, look at the following business transaction:

A customer of a travel agency wants to book a trip. This consists of booking a hotel, booking a flight, and booking the transport to and from the airport. If any of those bookings fails, the other ones become worthless, so it is apparent, that this is a business transaction.

But for this process, we do not necessarily have to fulfil ACID. It is enough to cancel the booked flight, and the taxi to and from the airport, if the booking of the hotel fails. Same for the other possible combinations.

Scenarios like this are so common, that BPMN (Business Process Model and Notation) [Mod11] has elements to model transactions with compensations.

These elements are supported by many business process engines, for example Camunda [22a].

3 Motivation and Related Work

In this chapter we want to illustrate the problem of nested transactions over multiple blockchains using an example scenario. We will also look into other works related to this problem.

3.1 Scenario

To understand the problem we want to solve, we first want to think about a possible scenario.

Let us imagine a manufacturer for some expensive, custom-made item. This manufacturer takes in orders from customers. Every item needs parts from different external suppliers. Shelve space is expensive, so everything is ordered on demand from those suppliers. When the ordered parts are received, the product is manufactured and a pick up notification is sent to the customer. Figure 3.1 shows a simplified illustration of this scenario.

But what if something goes wrong when ordering the parts? Maybe one supplier is out of stock. In this case the manufacturer now has to cancel the customers order, and notify them. Figure 3.2 shows the scenario with this case added.

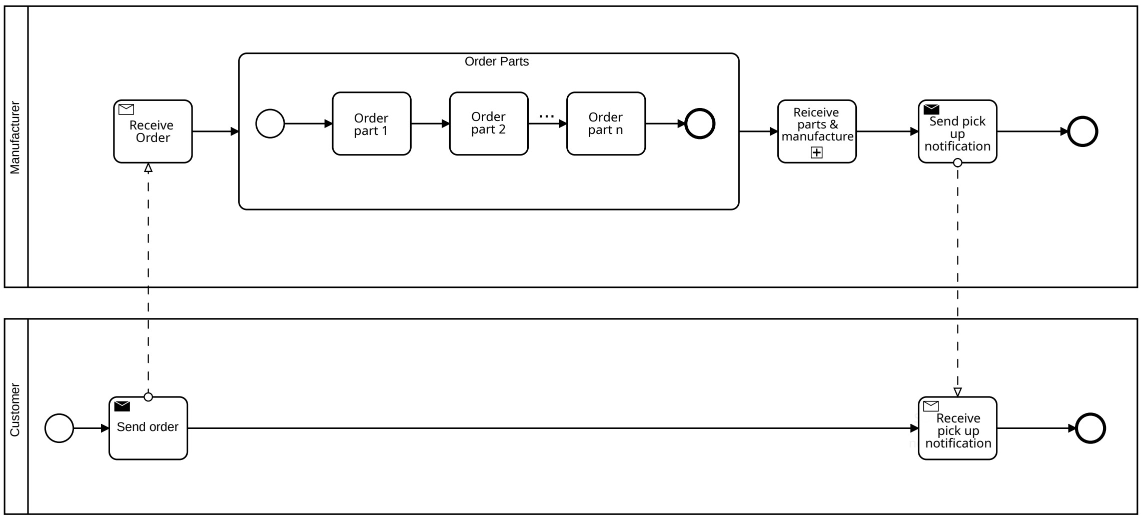

Figure 3.1: Illustration of our example scenario. A manufacturer receives an order, orders parts and then sends a pick up notification to the customer after manufacturing the ordered item.

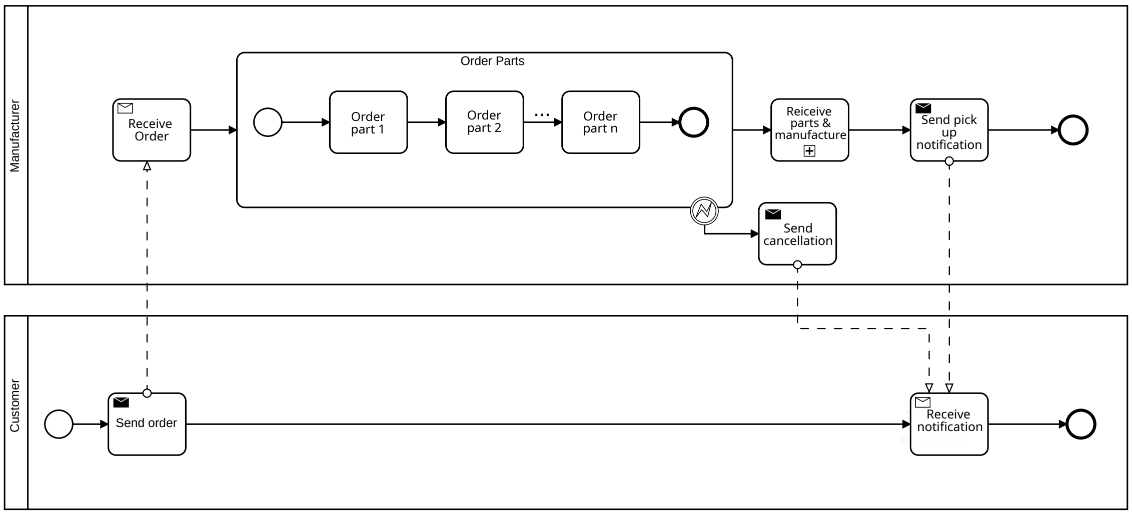

Figure 3.2: Illustration of our example scenario. A manufacturer receives an order, orders parts and then sends a pick up notification to the customer after manufacturing the ordered item. If the ordering of parts fails, the customers order is cancelled instead.

Is this really everything the manufacturer has to do in this case? Remember: the manufacturer orders parts from multiple suppliers. If one supplier can't deliver, this doesn't mean all can't deliver. But we assume, that the manufacturer doesn't have the space for storing parts (or just doesn't want to store parts, which might not be used soon). This means the manufacturer has to cancel all already sent orders.

It is easy to see the ordering of one part as a transaction, and likewise it is possible to understand the ordering of all parts as a nested transaction of those single orders.

If we can realize the ordering as a transaction, the whole cancellation will be automated by the transaction processing system. This saves the manufacturer from the complex and error-prone process of cancelling every order manually.

Let us further assume, that the external suppliers use smart contracts for implementing their ordering system. And let us also assume, that some of the suppliers use different blockchains than the others for their smart contracts.

With those assumptions the part ordering process isn't just a nested transaction, but also becomes a cross chain smart contract invocation.

This means we need a way to implement this cross chain smart contract invocation as a nested transaction.

3.2 Why Blockchain?

In the example scenario described in Section 3.1, we assume, that blockchains are used to manage the ordering process. But why should we use blockchains for this task at all?

In Section 2.1.3 we already saw the idea of using blockchains to implement smart contracts. It makes sense to use smart contracts in a scenario, where the participants don't want to rely on trusting the others. One of those scenarios is supply chain management.

Montecchi et al.[MPE19] state that using blockchains for supply chain management is a great way to increase the provenance knowledge of the customer. Provenance knowledge is the knowledge about origin, production and modification history and custody of a product.

By tracking the different stages of a product using smart contracts, one can provide the customer with assurance for origin, authenticity, integrity and custody, thereby reducing the perceived risk of the customer. And using blockchain technology for this, instead of e.g. certification by third party, can save the manufacturer money and decrease the complexity of the supply chain management. [MPE19]

One example is the Block-Supply Chain[AB18], a blockchain based system to fight counterfeiting of pharmaceuticals, proposed by Alzahrani and Bulusu. This system, which is a blockchain based version of an earlier, not blockchain based, proposal [AB16], uses Near Field Communication tags and a blockchain as database, to allow buyers to check the validity of the product they bought.

If we assume blockchain based supply chain management for the parts in our example (see Section 3.1), and we assume that we interact with different supply chains for each part, we easily get a scenario where we have to interact with smart contracts for the ordering process.

3.3 Transaction Semantics of Smart Contract Invocations on a single Blockchain

Before looking at the transactional behaviour of cross chain smart contract invocations, it makes sense to look at a single smart contract on one blockchain.

For a single blockchain, the different transactional properties can be found out by looking into the specifications of the blockchain.

Let us first look at permissionless blockchains. We concentrate on Ethereum like blockchains.

3.3.1 Transaction Semantics of Permissionless Blockchains

A blockchain is a distributed system. So a transaction consisting of smart contract invocations is a distributed transaction. Since the smart contract execution happens on a single system, during the state transition from the old to the new block, it still makes sense to look at the ACID properties of this local execution first.

Atomicity: A smart contract is executed, as part of the state transition. If an error occurs during execution of the smart contract, the execution is stopped, and all state changes get reverted, except for paid fees. [But+14]

If we look only at the execution of the code, and the changes made explicitly by the code, this means we have atomicity.

For most cases this behaviour is close enough to atomicity, but it doesn't hold if we look at the whole system. The smart contract execution does change the state of the system, even in case of errors, in the form of fees, which technically violates atomicity.

Consistency: Transactions in a block are validated and acknowledged by each node in the blockchain network, and only a valid block is accepted by the network.[TEK17] This means that the state before the execution (the last block), and the state after the execution (the new block), are both consistent from the consensus protocol's perspective. This means the execution of the smart contract will execute correctly. The responsibility for upholding any further consistency constraints is up to the smart contract programmer.

Isolation: Blockchain transactions have a well defined order [TEK17].

In Ethereum like blockchains, for each blockchain transaction of the new block a state transition function is performed. Smart contracts are executed during this state transition [But+14], This means they are executed in sequence, so parallel executions on one node don't exist, and cannot interfere with each other. Furthermore this sequence is the same on every node.

This means Isolation is fulfilled.

Durability: Proof of Work blockchains, like Ethereum and Bitcoin, do not reach finality.[TEK17]

Ethereum runs on a peer to peer protocol. If a new block is mined by a node, it announces it to the network. Every node that receives this announcement, validates it and propagates it to more peers. [KSL+21] With this the knowledge about the block spreads through the network.

If a block is valid depends on different criteria. For example a valid block has an existing and valid parent block and the block is newer than the previous block. [But+14] Another criteria is a valid proof of work. To create, or mine a new block, a node must create a proof of work. This is a nonce, that together with the block creates a hash that is smaller than a given number. This process it time and energy consuming, but easy to verify.

It is possible that multiple miners create a new block around the same time. Because of this and the time it takes to propagate the knowledge of blocks through the network, it can happen, that different branches of the chain emerge. This is called chain fork.

Per definition the branch, that has accumulated the most work in form of blocks, is valid. And every honest miner node will work on growing the valid chain. If a node learns of another branch that has accumulated more work and is therefore longer than its current valid one, it will update its knowledge and assume the longer chain as the valid one.

This means, that a block can become invalid, and with this the results of executed smart contracts on this chain.

If the majority of nodes is honest, and has the majority of computational power, this will lead to the valid chain growing faster than the invalid ones. This means that with every new block, it becomes more and more improbable, that a previous block in this branch gets invalidated.[But+14]

This means instead of durability, we have a probabilistic measure of finality. [TEK17].

While this is true for proof of work blockchains, there is also another consensus algorithm used by permissionless blockchains, called proof of stake.

In fact Ethereum is planing to switch to a proof of stake consensus algorithm. And in switching to the new system they propose a new protocol [BG17], where checkpoints are voted by validators. Those checkpoints give us a better guarantee of finality. Every 100 blocks a new checkpoint is created, and the state of the chain before this checkpoint cannot change anymore.

This new protocol might give us some real durability instead of a probabilistic finality.

As a side note: Since a certain amount of Ether, the cryptocurrency of the Ethereum blockchain, is needed to become a validator, one could argue, that the new role of validators would no longer make Ethereum a permissionless blockchain. Instead, it could be called some kind of hybrid. While this point is valid, it should be kept in mind, that becoming a miner in a proof of work chain isn't free as well, since it requires real world hardware. This makes both approaches not that different in this regard.

Now let us look at transaction criteria for distributed transactions.

Global atomicity: Global atomicity is the criteria, that if a transaction commits or aborts on one node, it does the same on all nodes of the system.[FKBL20] That means that the transaction will either completely execute or does not execute at all, and that consistently on all nodes.

Because of the possibility of chain forks, this is not given for permissionless blockchains. But in the same way as the durability, the probability that this is fulfilled increases with time.

Global isolation: A distributed system should provide the same level of isolation as a non-replicated system. This is for example given, if the execution over the distributed, replicated system is equivalent to an execution over a single replica.[KJPA10]

Again this is not given for permissionless blockchains, because of the possibility of chain forks. But if we ignore chain forks, and assume all nodes assume the same history, this is fulfilled, as the execution order is the same for a block, independent of the node. This again means, that the probability for this criterion increases with time.

Session consistency: Session consistency is a correctness criterion from the perspective of the user. If the user sends two transactions T_1 and T_2 in this order, then they expect that if T_1 writes into data, that is read by T_2 that T_2 will read the value T_1 has written, and not the state before T_1 updates it. This should hold on every node of the distributed system. [KJPA10]

This session consistency is not given for public blockchains. There is no guarantee that blockchain transactions get accepted by miners in a given order, so there is no guarantee for the execution order of smart contracts.

3.3.2 Transaction Semantics of Permissioned Blockchains

There is a large variety of different permissioned blockchains, with different capabilities and different consensus mechanisms. This variety leads to different transaction processing characteristics.

Falazi et al. [FKBL20] have analysed different permissioned blockchains and their transaction properties according to 4 correctness criteria: Local ACID, global atomicity, global isolation and session consistency. They found that the analysed blockchains fall into three categories: 1. Systems based on public blockchain systems, that may suffer from chain forking. Those do not fulfil any of the 4 criteria, and are comparable to their permissionless ancestors. 2. Systems that only lack session consistency. 3. Systems that fulfil all 4 criteria.

While it depends on the details of the blockchain, permissioned blockchains can give us better transactional guarantees than their permissionless counterparts.

3.4 Transaction Semantics of Cross Chain Smart Contract Invocation

We have seen that the transaction semantics of a smart contract on a single blockchain are clear. But what about the execution of multiple smart contracts?

In the simplest case the smart contracts live on the same chain and are invoked using a smart contract on this chain. On the Ethereum blockchain, for example, a smart contract can call functions of other contracts. The execution of such a nested smart contract invocation plays out as if it was a single smart contract. [But+14]

But what if the smart contracts involved are running on different blockchains? What about cross chain smart contract invocation?

In this case the transaction semantics are not clear. It isn't even clear how the smart contracts are called.

There are several problems:

1.How to invoke the smart contracts? While a single blockchain might have a way to call smart contract functions on the same blockchain, there is no unified method to call a smart contract on another blockchain from within a smart contract. Without a method to actually invoke cross chain smart contracts, nested transactions consisting of smart contracts are not possible.

We need a method to formulate and execute a nested transaction consisting of smart contracts.

2. How should those smart contracts communicate? The nature of blockchains makes it difficult to get information outside the blockchain. E.g. an Ethereum smart contract is executed, when a peer validates its block. We have no guarantees about when that happens. In such a system, calls to outside the blockchain are difficult. There are concepts like oracles [22b], that use an off-chain service, that monitors the blockchain for a request on which it writes data to the blockchain, that then can be read by another smart contract function. Such a communication needs multiple steps.

So if we want to have communication between smart contracts in a cross chain smart contract invocation, we will need some middleware to handle the communication between the smart contracts. If this middleware also handles the invocation, the middleware could use input parameters for communication, otherwise it has to write the information to the blockchains.

3. What transaction properties are possible? In case of nested transactions, we want to provide transaction guarantees as well.

We would want to keep as much of the single blockchains transaction guarantees as possible. So lets look at the transaction guarantees one at a time.

Atomicity: Ideally we want atomicity for the whole nested transaction. This means if one smart contract fails, none of the other smart contracts should change the state of its blockchain.

A big problem for this is the fact, that a smart contract cannot know during execution, if a parallel or later executed smart contract on another chain executed or aborted. We saw earlier, that the communication between chains needs multiple smart contract invocations, and changes to the blockchain state. So we cannot guarantee atomicity for the whole blockchain state.

If we only look at the state of the smart contract, one could implement protocols similar to traditional database systems, that only commit the changes, when all participating smart contracts commit them.

Consistency: Assuring consistency is a problem as well. For example, assume we have a token transfer (see Section 2.1.4) implemented using smart contracts over two different blockchains A and B. If A sends the token to B, and the smart contracts on any sides are forgotten (e.g. because of chain forks) we can end up in situations where the token is not removed from blockchain A, leading to duplication of the token, or a situation where the token is deleted from chain A, but the smart contract execution to receive it on blockchain B is forgotten, leading to the destruction of the token. Traditional databases solve this problem with protocols like 2 Phase Commit, but it is unclear if implementing existing protocols is enough for cross chain smart contract invocations, especially when dealing with uncertain durability.

Isolation: Similar to the problem of atomicity, isolation can be easily violated in cross chain smart contract invocations. Assume we have a cross chain smart contract invocation that communicates between two blockchains:

$$SC_1, SC_2, SC_3$$

where SC_1, and SC_3 lie on blockchain A and SC_2 lies on blockchain B. In the time between the end of the execution of SC_1 and the execution of SC_3 (which happens after the execution of SC_2 on another blockchain), we have an intermediate state on blockchain A. Other than in the single blockchain case, we have to explicitly handle isolation inside the smart contracts of a cross chain smart contract invocation.

Durability: Durability was already a problem in the single blockchain case. If we assume a probabilistic finality for single smart contract invocations, we will only be able to give a probabilistic guarantee for cross chain smart contract invocations.

3.5 Related Work

There are multiple works in the field of blockchain interoperability.

Hermes [BVCH21] is a middleware that uses a protocol to guarantee ACID for cross chain blockchain transactions. For this it utilizes gateways, that can for example be implemented as smart contracts on the connected blockchains. They mainly concentrate on asset transfer, and not on the invocation of smart contracts and the state of their execution.

Polkadot [Woo16] is a multi-chain approach, where a blockchain called relay chain allows for the sending of blockchain transactions between so called parachains. They use nodes with different roles to allow for trust free communication between different blockchains. The most important roles are validators and collators, which work together. In short the collators execute transactions on the parachain and send blocks with proof to the validators. The validators then validates those candidate blocks. The integration of existing blockchains like Ethereum might be possible using transaction forwarding contracts, but is complicated. The cross chain blockchain transactions are asynchronous and one way, meaning there is no way to directly return an answer between the parachains. With this the invocation of smart contracts on another chain should be possible, but it does not solve the problems stated in Section 3.4.

A similar approach is Cosmos[KB19], which uses Hub blockchains to allow communication between other blockchains called Zones. It concentrates on asset exchange, but smart contract capable Zones should also be possible. It uses Tendermint[Buc16] as consensus mechanism for both Hub and Zones. To add non Tendermint blockchains, one would need a Zone acting as adapter. While not the direct goal of Cosmos, one might use it to implement a transaction middleware for heterogeneous blockchains in the future.

Liu et al. propose HyperService [LXS+19], a platform for programming distributed applications over heterogeneous blockchains. It allows for cross chain smart contract invocation. They propose a model for describing the applications, and a cryptographic trust free peer to peer protocol, to invoke the smart contracts. They use peers that act as gateways to other blockchains. HyperService only works between blockchains with a public transaction ledger (so many permissioned blockchains aren't compatible) Instead of ACID atomicity, HyperService guarantees what they call financial atomicity, in which they guarantee, that the cost of running a failed cross chain smart contract invocation is recompensated. They do not revert state changes done by partly executed cross chain smart contract invocations though. To handle cases, where the state would need to be reverted, they propose the concept of stateless smart contracts. These stateless smart contracts are smart contracts, that can use the state of an older block, to solve this issue. But this idea is not yet found in any blockchain implementation.

Other works do not try to solve the problem of blockchain interoperability, but try to connect blockchains with other technology.

BlockME[FHBL19] is an extension of BPMN, that enables to integrate blockchain transactions into BPMN Processes. BlockME consists of several new tasks to communicate with different blockchains in an unified way, using an extensible gateway called Blockchain Access Layer. BlockME2[FHB+19] even introduces a task for invoking smart contracts. While BlockME handles several error scenarios, it does not explicitly look into transactions or cross chain scenarios.

SCIP [FBD+20] is a protocol that allows to invoke and query smart contracts of different blockchains in a blockchain independent way.It handles durability of smart contract invocations by allowing the user to request a degree of confidence for the smart contract invocation to not be revoked in the future, and a time limit in which this degree of confidence has to be reached. If it isn't reached it throws an error. SCIP works with single smart contract invocations on single blockchains, and has no support for cross chain smart contract invocations.

4 Proposal

In this chapter we will propose using the saga pattern as a solution to the problem stated in Chapter 3.

4.1 Using the Saga Pattern for Cross Chain Smart Contract Invocation

In Section 3.4 we have seen the challenges a system for nested transactions consisting of cross chain smart contract invocations faces.

If we want to give strong guarantees, we would need to implement a complex protocol, between the different blockchains, and make strong assumptions about the smart contracts involved.

For example, if we would want to implement ACID guarantees for certain variables in the contract, we would need to implement a locking mechanism for those resources, on each blockchain participating, and rewrite every smart contract participating potentially heavily.

Also locked resources might be locked for a long time, since blockchains are relatively slow, and we have to wait for a certain time, to get to a certain degree of confidence for durability.

If we look back at our example from Section 3.1 this would mean, that we need to get every supplier, to modify their contracts to use the same protocol. And they must make sure, that a potential long locking of resources doesn't hinder their business.

This is an unrealistic requirement.

Our part ordering example actually looks similar to the travel agency example from Section 2.2.2. In both cases we have several parties, with potentially individual booking/ordering processes. And each of those processes might take a relative long time. The booking example was an example for a business transaction, in which the saga pattern, and it's reduced transaction guarantees, where sufficient.

If we apply the saga pattern, there is no need for a unified protocol between all participants. We only need some way to cancel the orders as a compensation, which in most ordering scenarios is already given.

We would also not lock resources over a long time, so no changes needed on the suppliers side for this.

To use the saga pattern, we need a way to track the state of the execution of the saga. This is needed to know which compensations need to be executed in case of error.

We also need some way to describe the saga, and some way to communicate with the different blockchains.

Ideally the used methods are easily integrated into existing processes.

As solution, we propose an extension of BPMN, similar to BlockME [FHBL19], to utilize the existing saga support of business process engines.

For the communication with the blockchains, we will use SCIP [FBD+20], which gives us a unified way to invoke blockchains on different blockchains.

4.2 Saga Model for Cross Chain Smart Contract Invocation

Similar to the formal definition of a saga for transactions by Garcia-Molina and Salem (see Section 2.2.2) we want to define a saga of smart contract invocations as a sequence of smart contract invocations

$$SC_1, SC_2 ..., SC_n$$

and corresponding compensating smart contract invocations

$$C_1, C_2, ..., C_n$$

where C_i is a smart contract invocation that compensates the effects of SC_i. And probabilities

$$P_{SC_1}, ..., P_{SC_n}, P_{C_1}, ..., P_{C_n}$$

one for each invocation, that states the desired minimal probability of its invocation not being undone in the future.

For the saga it is guaranteed that either the sequence

$$SC_1, SC_2 ..., SC_n$$

or the sequence

$$SC_1, SC_2, ..., SC_j, C_j^m, C_{j-1}^m, ..., C_1^m \quad (0 \leq j \leq n)$$

where

$$C_i^m = \underbrace{C_i, C_i, ..., C_i}_{1 \geq n \leq m \text{ times}}$$

or nothing at all is executed.

For the executed sequence, each SC_i has a probability of at least P_{SC_i} to not be undone, and for C_i^m the last C_i has a probability of at least P_{C_i} that its execution will not be undone in the future. (Note that in this context compensating a smart contract invocation with another smart contract invocation does not qualify as the execution being undone.)

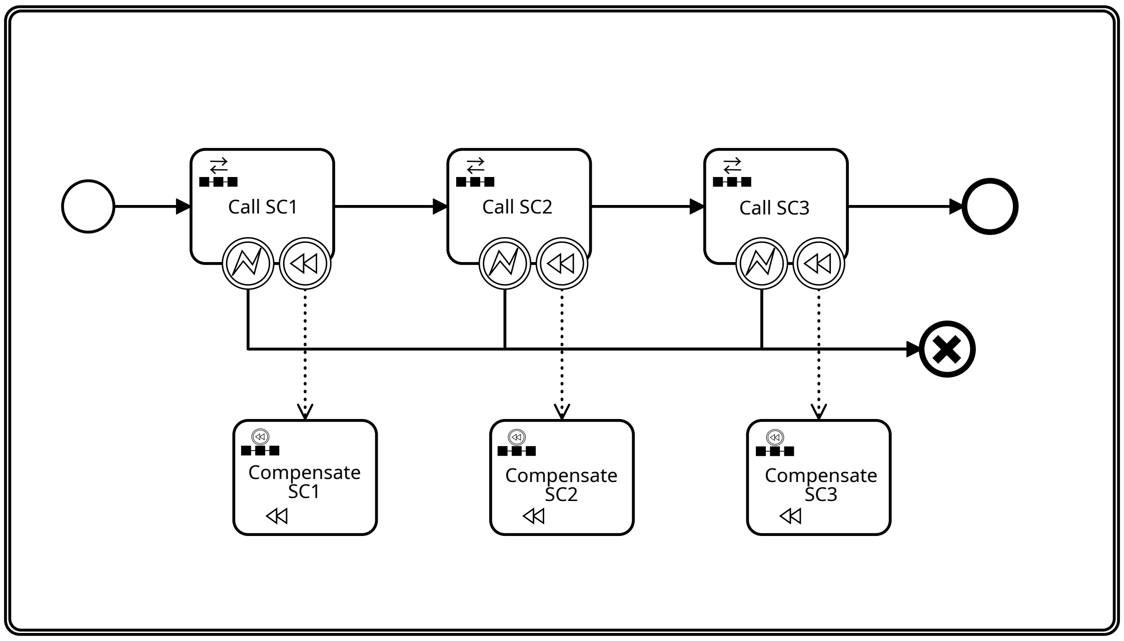

Figure 4.1: Transaction utilizing our BPMN extension in untransformed state

We want to allow for multiple invocations of C_i for technical reasons. We might have cases, where we are not able to guarantee a successful execution of a compensation invocation, but also can not guarantee, that it isn't executed. One example is the case, where the corresponding blockchain transaction reaches a block (so the contract is executed) but we might not be able to guarantee the desired probability in a reasonable timeframe. (We can not know if the desired probability will ever be reached, so an implementation will have to give up waiting after a certain amount of time. An example for this is the Timeout error in SCIP. See Section 4.3.1.). But since we need a succesful invocation of C_i to undo the effect of SC_i, we allow for a retry of the invocation, possibly leading to multiple invocations of C_i.

Those cases are also the reason, why we define a compensation for SC_n. If SC_n does not reach the desired probability, it should be compensated as well.

The parameter m will be customizable by the user in case multiple executions are not wanted (e.g. non idempotent compensation contracts).

4.3 Extending BPMN

To allow for easy modelling of sagas consisting of smart contracts, we propose an extension of BPMN. This allows for easy integration into existing processes.

Our extension consists of 2 new types of BPMN tasks.The Smart Contract Invocation Task, and the Smart Contract Compensation Task. Those two tasks should be used inside of BPMN transactions. Figure 4.1 shows an example of a transaction, using the untransformed tasks.

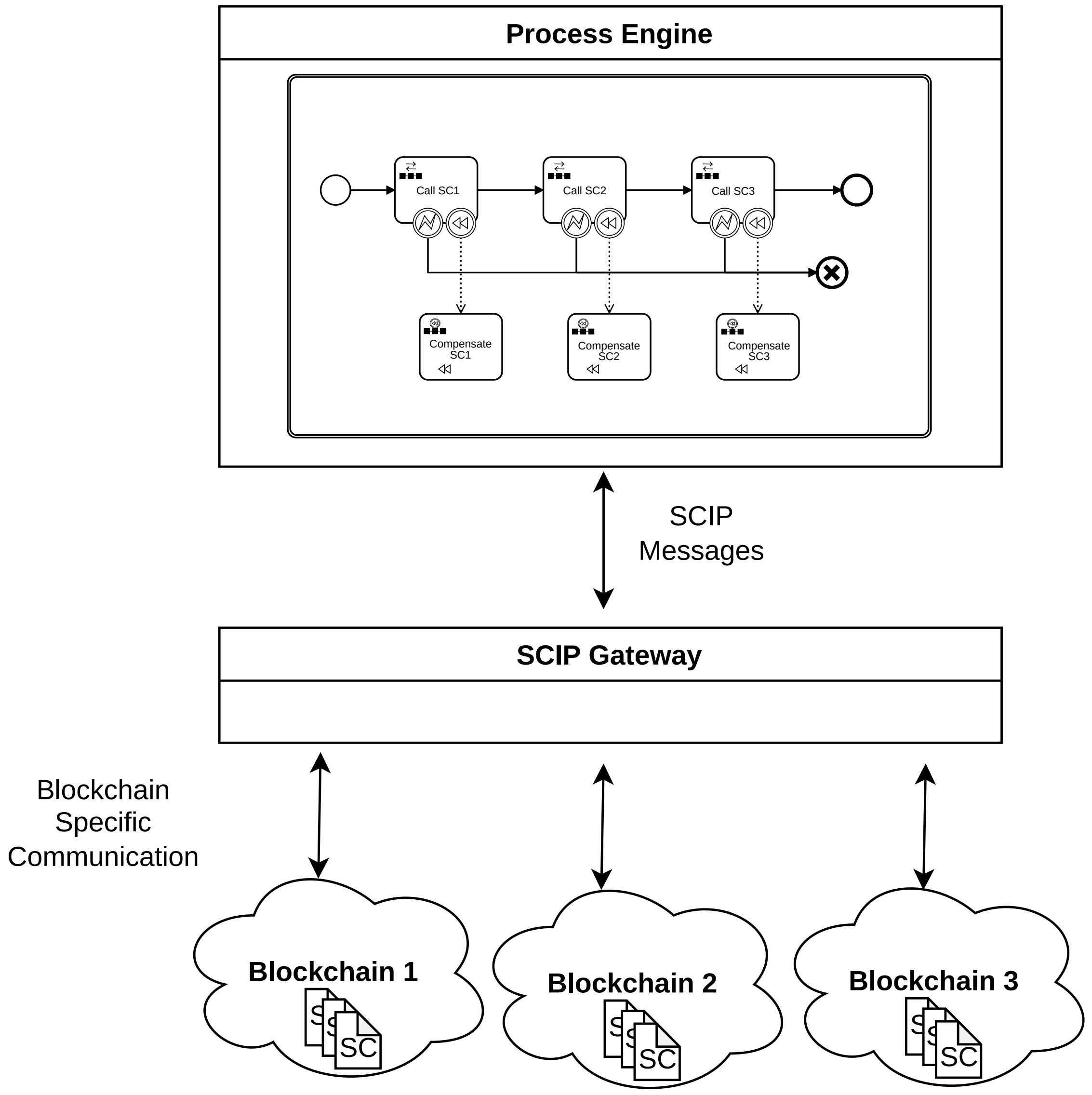

Figure 4.2: Overview over involved parts

Before deployment into a process engine, a BPMN model that uses the tasks has to be transformed into standard BPMN. Figure 4.3 and Figure 4.4 show the transformations needed. We will describe the transformations in detail later in this chapter.

Both tasks make use of SCIP to invoke the smart contract, get potential return parameters, and to track potential errors.

Figure 4.2 shows an overview of the resulting architecture. We have a process engine running our business process, including a transaction with our extension. The process engine calls a SCIP gateway using SCIP. The SCIP Gateway then handles the communication with the different blockchains.

4.3.1 SCIP

SCIP(Smart Contract Invocation Protocol)[FBD+20] is a protocol that provides a unified way to invoke smart contracts. For this it uses gateways, which are connected to one or more blockchains, and are able to communicate with them.

SCIP provides four methods to interact with smart contracts, invocation, subscription, unsubscription and query. The subscription method subscribes the caller to notifications about invocations and other events regarding a smart contract, the unsubscribe method unsubscribes the caller from those notifications. The query method allows to query for past events and invocations.

The invocation method invokes a smart contract function.

We only use the invocation method. It has the following parameters:

- Function identifier: The name of the function to invoke
- Inputs: A list of input parameters for the function
- Outputs: a list of output parameters, expected of the function
- Callback URL: location where asynchronous callbacks should be sent to.
- Correlation identifier: Identifier to correlate callback and invocation
- Degree of confidence: The probability wanted, that the smart contract's blockchain transaction is final, to consider it correct.
- Timeout: How long to wait for the blockchain transaction to gain the wished degree of confidence, before considering it failed.
- Signature: base 64 encoded signature of the request

The invocation request is sent to a Smart Contract Locator[LFB+20] URI of the gateway. The SCL holds information like the address of the smart contract, and the blockchain used.

The degree of confidence parameter is SCIPs solution to the problem of finality or durability of a single contract invocation (see Section 3.3.1). It allows the user to state which probability of finality is enough for them, to consider the execution final.

SCIP's invocation method gives us two kinds of replies, a synchronous reply (OK or error), and an asynchronous callback. Both can contain an error message, with an error code. If successful, the asynchronous callback contains the smart contracts return values. Table 4.1 shows some example errors and error codes.

4.3.2 The Smart Contract Invocation Task

The first of our two new BPMN tasks is the smart contract invocation task. See Figure 4.3 for its untransformed representation as well as its transformation to standard BPMN.

This task represents a smart contract invocation, inside a saga consisting of smart contracts.

Error Code	Error Name	Description
-32001	InvalidParameters	The input parameters do not match the function.
-32007	InvalidScipParam	A SCIP parameter has an invalid value
-32100	InvocationError	General error when trying to invoke the smart contract function. Specific cause is unknown.
-32101	ExecutionError	An exception was thrown by the smart contract.
-32201	Timeout	Timeout reached before the desired degree of confidence was reached.

Table 4.1: A list of SCIP errors. For a full list see [19].

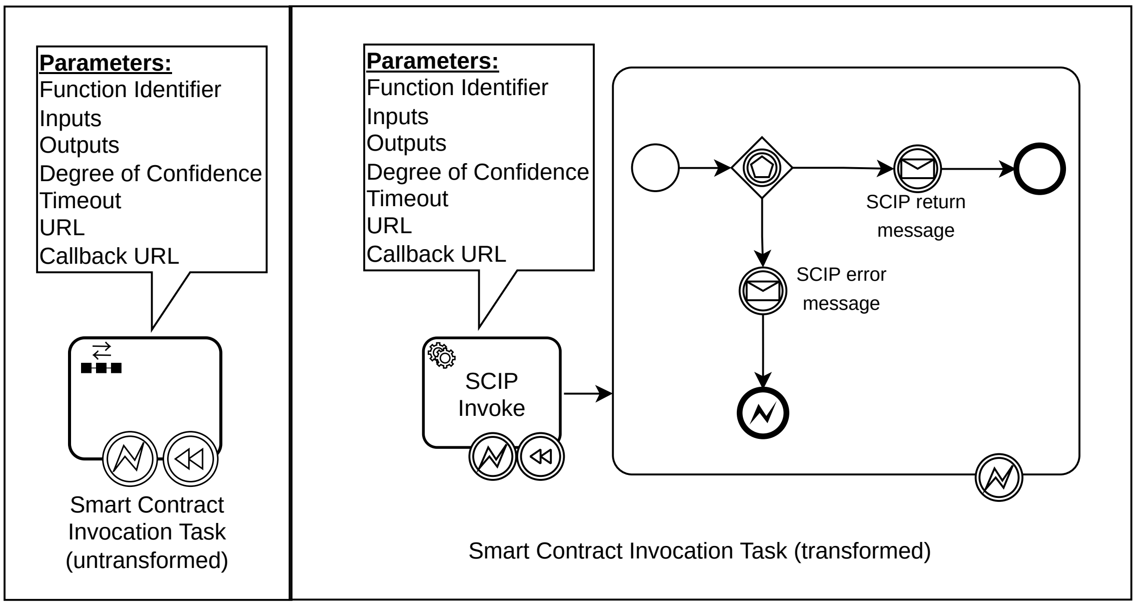

Figure 4.3: Transformation of the smart contract invocation task. See Section 4.3.2 for a description of the task and its parameters.

Its has the following user defined input parameters:

- Function identifier: The name of the function to invoke
- Inputs: A list of input parameters for the function
- Outputs: a list of output parameters, expected of the function
- Degree of confidence: The probability wanted, that the smart contract's blockchain transaction is final, to consider it correct.
- Timeout: How long to wait for the blockchain transaction to gain the wished degree of confidence.
- URL: the SCL of the smart contract
- Callback URL: The address of the process engines message endpoint

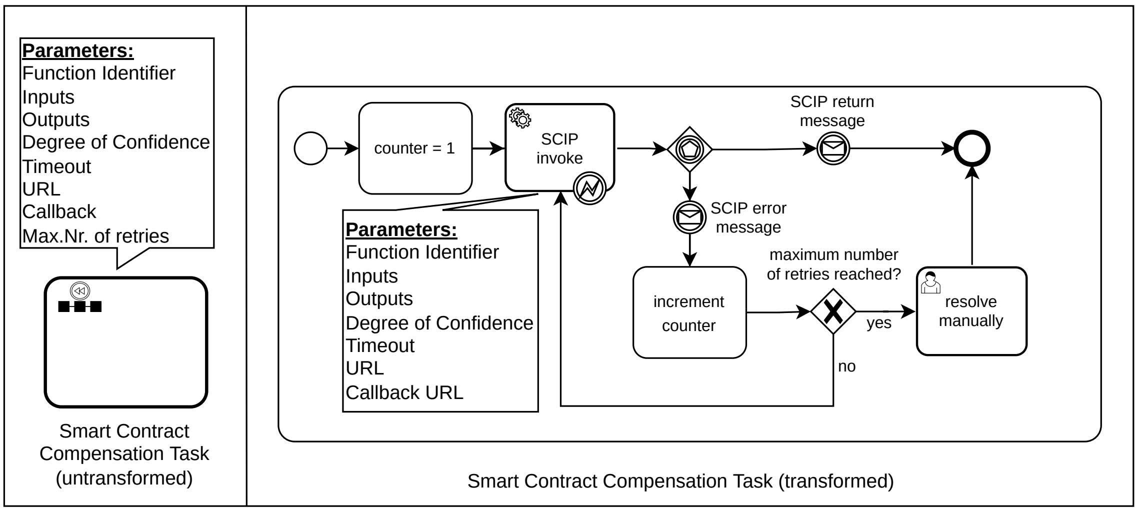

Figure 4.4: Transformation of the smart contract compensation task. See Section 4.3.3 for a description of the task and its parameters.

The parameters have the same purpose and format as the corresponding SCIP parameters.

The smart contract invocation task has two boundary events, an error boundary event and a compensation boundary event. The error boundary event is triggered, when the SCIP invocation returns an error.

In transformed form, the task invokes the given smart contract function using SCIP. If the SCIP invocation throws no synchronous error, the task awaits the callback. If the callback message is a successful return message, the task ends. Potential return values are sent with the return message. If either a synchronous error is thrown or the asynchronous callback is an error message, the whole task throws an error.

The transformed task is split into two, because we also want to trigger a compensation for the last task in the saga, if it suffers from an asynchronous error like a timeout. The error boundary events of both tasks should be connected to the cancelation event of the transaction.

4.3.3 The Smart Contract Compensation Task

The second of our two new BPMN tasks is the smart contract compensation task. See Figure 4.4 for its untransformed representation as well as its transformation to standard BPMN.

The smart contract compensation task acts similar to the smart contract invocation task, but it is used as compensation for a smart contract invocation in a saga.

It has the same parameters as the smart contract invocation task, plus the *maximum number of retries*.

As a compensation task, it has no boundary events.

In transformed form the task first initializes a counter. It then tries to invoke the corresponding smart contract function using SCIP. If no synchronous error is thrown, the task waits for the callback message. In the same way as the smart contract invocation task, the task distinguishes return and error callback messages. If a synchronous or asynchronous error is encountered, the counter is incremented. If it is smaller than the maximum amount of retries, another attempt at invoking the smart contract function is done. If the maximum number of retries is reached, the compensation is handed to a human to resolve manually.

The differences between this and the smart contract invocation task lay mainly in the retry mechanism. If the smart contract invocation task fails, it is intended, that the corresponding compensation task is triggered. For the compensation task this is not possible. Instead, we involve a human, but this is an unwanted scenario. To minimize the need of human intervention, the retry mechanism was added. It might solve the problem in cases like timeouts when waiting for the requested degree of confidence, or when the transaction simply does not make it into any block, because of a temporary spike in blockchain activity.

If the smart contract function used for compensation is not idempotent, it might be a good idea to set the maximum retry count to 1. The reason for this is, that if the smart contract invocation returns a timeout error, this does only mean that the required degree of confidence wasn't reached in time. The smart contract might still be executed. This could be problematic for non idempotent compensation smart contract functions.

As a compensation task, the smart contract compensation task has no error boundary event.

5 Implementation

As a proof of concept, we implemented the proposed extension using Camunda Platform [22a] as business process engine and the Blockchain Access Layer(BAL) [FHBL19] as SCIP Gateway.

The proof of concept can be found at https://github.com/dabberpk/CaseStudy

5.1 Blockchain Access Layer

BAL[FHBL19] is an abstraction layer that offers a uniform interface for blockchains. The communication with the different blockchains is handled by blockchain specific adapters. It also implements a SCIP endpoint, using a JSON-RPC API.

We use a feature branch[1] of BAL that allows for choosing the message format of the SCIP callback message. We use this to ensure that Camunda understands the format of the callback messages.

5.2 Camunda

Camunda Platform [22a] is an open source workflow engine that allows for the execution of BPMN 2.0 processes. It exposes a REST API, which can be used to send messages to the processes.

Camunda Platform allows for BPMN service tasks to be implemented in Java, which we use to make it communicate with SCIP.

5.3 Connecting Camunda and SCIP

The communication between SCIP and Camunda happens in two ways. First there is a service task implemented as a Java delegate. This Java delegate takes inputs we already have seen in Section 4.3.2 and converts them into a JSON-RPC request for the SCIP Gateway. It then sends the JSON-RPC request message as a POST request to the SCIP Gateway and awaits the reply. If the reply is an error message, the delegate throws a BPMN error that can be caught by the BPMN process.

The code of the java delegate can be found at https://github.com/dabberpk/SCIP-Connector

[1]https://github.com/TIHBS/BlockchainAccessLayer/tree/feature/enable-choosing-bindings

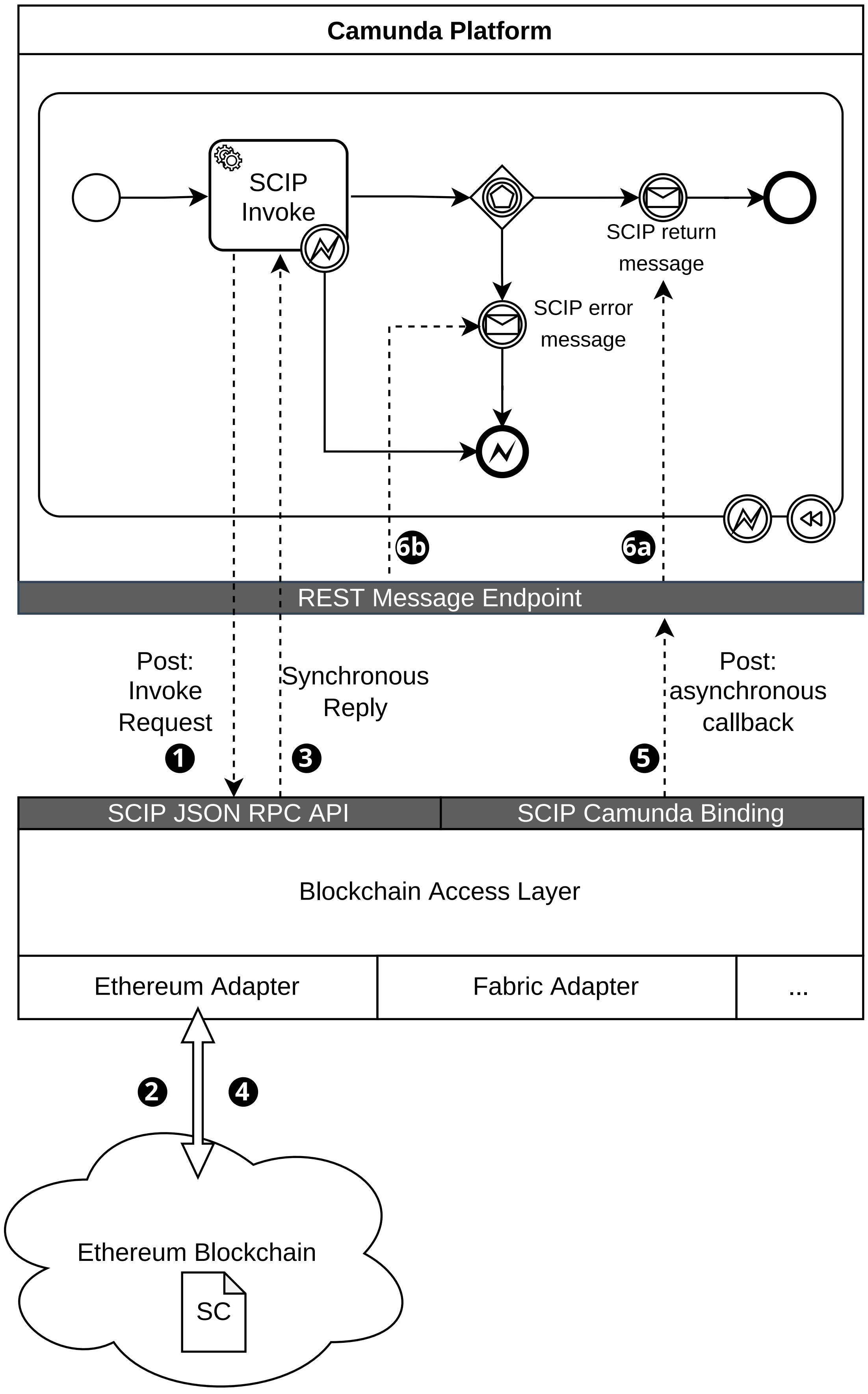

Figure 5.1: Overview over the parts involved in the proof of concept and their communication

5.4 Implementation Overview

Figure 5.1 gives an overview over the involved parts and how they communicate during an invocation.

Camunda Platform is used as BPMN process engine and runs a transformed BPMN model. The model task uses the SCIP Connector Java delegate, to send a POST request with the JSON RPC request message to the SCIP JSON-RPC API of the Blockchain Access Layer (1). BAL then sends a blockchain transaction to the blockchain using the corresponding adapter (2). Now BAL answers the POST request with the synchronous JSON-RPC reply (3). If the invocation failed at this point the answer will contain an error, and the Java delegate throws an BPMN error. Otherwise the process goes on and awaits an asynchronous callback message. BAL periodically queries the blockchain for the status. Eventually, when the wanted degree of confidence for the invocation is reached, the timeout happens before this, or another error happens, the asynchronous callback is sent as POST request to the REST Message Endpoint of Camunda (5). The message is then correlated with the process and delivered to the corresponding message event(6a and 6b), and the business process goes on.

6 Conclusion and Outlook

In this thesis we have taken a look at the problem of transactions consisting of cross chain smart contract invocations. We first looked into the transaction properties of smart contract invocations on single blockchains, both permissioned and permissionless. We then saw that the same guarantees aren't given for smart contract invocations over different blockchains.

To tackle this, we introduced a modified saga pattern as a possible solution for transactions consisting of cross chain smart contract invocations. For this end we proposed an extension for BPMN that utilizes the SCIP protocol, for invoking smart contracts. This allows us to use existing business process engines and their implementation of traditional sagas with smart contracts.

The resulting saga model gives us similar properties as the original saga pattern, albeit with only probabilistic guarantees, inherited from the only probabilistic durability of the involved blockchains.

6.1 Advantages and Disadvantages

Using sagas for cross chain smart contract invocations has advantages as well as disadvantages.

The biggest advantage of our proposal is its flexibility. There are nearly no prerequisites to be fulfilled by the involved smart contracts. There only has to be a way to compensate them. There are no libraries that have to be used, and no protocols that have to be implemented. There are no changes necessary to the underlying blockchain, and there are no assumptions made about the location of the compensating smart contract function. It could be implemented in another smart contract. This means that many smart contracts, even already deployed ones, should be compatible without changes.

The fact that BPMN Processes are used leads to easy integration into existing processes.

While we have only looked at compensating smart contract invocations, the modeler is not forced to use smart contracts for compensating. In the same way it is possible to add other, non smart contract invocation tasks into the process. While this is a possibility, one should keep in mind, that it is unclear if this has any effect on the transaction properties of the whole saga.

Of course the proposed approach also has drawbacks. While there are not many prerequisites, the transaction guarantees given are weak as well.

Not only does this approach inherit the problems of the saga pattern, like missing isolation and atomicity for the overall transaction, it also has more problems coming from its differences to the original saga pattern. The fact that the single smart contract invocations themselves are not fulfilling ACID transaction guarantees makes the guarantees for the whole saga even weaker. We can only give a probability for the durability of a single invocation, and this not only leads to a probabilistic durability for the whole saga, but if a “successful” invocation later vanishes, this leaves the whole saga in an unclear state. If a high enough degree of confidence is demanded by the modeler, this should be highly unlikely, but it should still be kept in mind. Fortunately, there are blockchains where the durability isn’t that problematic.

Another disadvantage is, that with using process engines and gateways, we introduce a centralized element into the cross chain smart contract invocation. But as they can be self-hosted, the resulting trust issues can be mitigated.

6.2 Future Work

Blockchain interoperability and cross chain smart contract invocations is a young research field.

There are several directions one could go to try improve our approach.

While using the SCIP gateway makes it possible to easily integrate our saga model with existing technology, it also introduces a centralized component. It is thinkable to implement the saga model, using a blockchain with adapters as middleware, similar to how HyperService[LXS+19] works. But instead of guaranteeing financial atomicity, implement compensating transactions. Such a system would solve the centralization problem. It is questionable if any existing approaches could be modified for such a system, or if a new one has to be created.

Another research direction is, to try to aim for stronger guarantees. One could try to adapt other known traditional mechanisms, like resource locking or protocols like 2 Phase Commit, to the blockchain context. Such an approach would probably result in stronger assumptions about the participating smart contracts, leading to less flexibility. But the gained assurances could be worth it.

Bibliography

[19] *Smart Contract Invocation Protocol (SCIP)*. Nov. 7, 2019. URL: https://github.com/lampajr/scip (cit. on p. 34).

[22a] *Camunda Workflow and Decision Automation Platform*. Mar. 30, 2022. URL: https://camunda.com/ (cit. on pp. 18, 37).

[22b] *Oracles*. Mar. 30, 2022. URL: https://ethereum.org/en/developers/docs/oracles/ (cit. on p. 25).

[AB16] N. Alzahrani, N. Bulusu. "Securing pharmaceutical and high-value products against tag reapplication attacks using nfc tags". In: *2016 IEEE International Conference on Smart Computing (SMARTCOMP)*. IEEE. 2016, pp. 1–6 (cit. on p. 21).

[AB18] N. Alzahrani, N. Bulusu. "Block-supply chain: A new anti-counterfeiting supply chain using NFC and blockchain". In: *Proceedings of the 1st Workshop on Cryptocurrencies and Blockchains for Distributed Systems*. 2018, pp. 30–35 (cit. on p. 21).

[ABB+18] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, et al. "Hyperledger fabric: a distributed operating system for permissioned blockchains". In: *Proceedings of the thirteenth EuroSys conference*. 2018, pp. 1–15 (cit. on p. 15).

[BG17] V. Buterin, V. Griffith. "Casper the Friendly Finality Gadget". In: *CoRR* abs/1710.09437 (2017). arXiv: 1710.09437. URL: http://arxiv.org/abs/1710.09437 (cit. on p. 23).

[BN09] P. A. Bernstein, E. Newcomer. *Principles of transaction processing*. Morgan Kaufmann, 2009 (cit. on pp. 16, 17).

[Buc16] E. Buchman. "Tendermint: Byzantine fault tolerance in the age of blockchains". PhD thesis. University of Guelph, 2016 (cit. on p. 27).

[But+14] V. Buterin et al. *Ethereum: A next-generation smart contract and decentralized application platform*. 2014 (cit. on pp. 15, 22, 23, 25).

[BVCH21] R. Belchior, A. Vasconcelos, M. Correia, T. Hardjono. "HERMES: Fault-Tolerant Middleware for Blockchain Interoperability". In: (2021) (cit. on p. 27).

[FBD+20] G. Falazi, U. Breitenbücher, F. Daniel, A. Lamparelli, F. Leymann, V. Yussupov. "Smart Contract Invocation Protocol (SCIP): A Protocol for the Uniform Integration of Heterogeneous Blockchain Smart Contracts". In: *Advanced Information Systems Engineering*. Ed. by S. Dustdar, E. Yu, C. Salinesi, D. Rieu, V. Pant. Cham: Springer International Publishing, 2020, pp. 134–149. ISBN: 978-3-030-49435-3 (cit. on pp. 11, 28, 30, 33).

[FHB+19] G. Falazi, M. Hahn, U. Breitenbücher, F. Leymann, V. Yussupov. “Process-Based Composition of Permissioned and Permissionless Blockchain Smart Contracts”. In: *Proceedings of the 2019 IEEE 23rd International Enterprise Distributed Object Computing Conference (EDOC 2019)*. IEEE, Oct. 2019, pp. 77–87. DOI: 10.1109/EDOC.2019.00019 (cit. on p. 27).

[FHBL19] G. Falazi, M. Hahn, U. Breitenbücher, F. Leymann. “Modeling and execution of blockchain-aware business processes”. In: *SICS Software-Intensive Cyber-Physical Systems* (Feb. 2019), pp. 1–12. ISSN: 2524-8529. DOI: 10.1007/s00450-019-00399-5. URL: http://link.springer.com/article/10.1007/s00450-019-00399-5 (cit. on pp. 27, 30, 37).

[FKBL20] G. Falazi, V. Khinchi, U. Breitenbücher, F. Leymann. “Transactional properties of permissioned blockchains”. In: *SICS Software-Intensive Cyber-Physical Systems* 35.1 (Aug. 2020), pp. 49–61. ISSN: 2524-8529. DOI: 10.1007/s00450-019-00411-y. URL: https://doi.org/10.1007/s00450-019-00411-y (cit. on pp. 23, 24).

[GR92] J. Gray, A. Reuter. *Transaction processing: concepts and techniques*. Elsevier, 1992 (cit. on p. 16).

[GS87] H. Garcia-Molina, K. Salem. “Sagas”. In: *ACM Sigmod Record* 16.3 (1987), pp. 249–259 (cit. on pp. 17, 30).

[KB19] J. Kwon, E. Buchman. “Cosmos whitepaper”. In: *A Netw. Distrib. Ledgers* (2019) (cit. on p. 27).

[KJPA10] B. Kemme, R. Jiménez-Peris, M. Patiño-Martínez, G. Alonso. “Database Replication: A Tutorial”. In: *Replication: Theory and Practice*. Ed. by B. Charron-Bost, F. Pedone, A. Schiper. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 219–252. ISBN: 978-3-642-11294-2. DOI: 10.1007/978-3-642-11294-2_12. URL: https://doi.org/10.1007/978-3-642-11294-2_12 (cit. on pp. 23, 24).

[KSL+21] L. Kiffer, A. Salman, D. Levin, A. Mislove, C. Nita-Rotaru. “Under the hood of the ethereum gossip protocol”. In: *International Conference on Financial Cryptography and Data Security*. Springer. 2021, pp. 437–456 (cit. on p. 22).

[LFB+20] A. Lamparelli, G. Falazi, U. Breitenbücher, F. Daniel, F. Leymann. “Smart Contract Locator (SCL) and Smart Contract Description Language (SCDL)”. In: *Service-Oriented Computing – ICSOC 2019 Workshops*. Ed. by S. Yangui, A. Bouguettaya, X. Xue, N. Faci, W. Gaaloul, Q. Yu, Z. Zhou, N. Hernandez, E. Y. Nakagawa. Cham: Springer International Publishing, 2020, pp. 195–210. ISBN: 978-3-030-45989-5 (cit. on p. 33).

[LXS+19] Z. Liu, Y. Xiang, J. Shi, P. Gao, H. Wang, X. Xiao, B. Wen, Y.-C. Hu. “Hyperservice: Interoperability and programmability across heterogeneous blockchains”. In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 2019, pp. 549–566 (cit. on pp. 27, 42).

[Mod11] B. P. Model. “Notation (BPMN) version 2.0”. In: *OMG Specification, Object Management Group* (2011), pp. 22–31 (cit. on p. 18).

[MPE19] M. Montecchi, K. Plangger, M. Etter. “It’s real, trust me! Establishing supply chain provenance using blockchain”. In: *Business Horizons* 62.3 (2019), pp. 283–293. ISSN: 0007-6813. DOI: https://doi.org/10.1016/j.bushor.2019.01.008. URL: https://www.sciencedirect.com/science/article/pii/S0007681319300084 (cit. on p. 21).

[Nak08] S. Nakamoto. “Bitcoin: A peer-to-peer electronic cash system”. In: *Decentralized Business Review* (2008), p. 21260 (cit. on pp. 13, 14).

[SSFB19] S. Schulte, M. Sigwart, P. Frauenthaler, M. Borkowski. “Towards Blockchain Interoperability”. In: *Business Process Management: Blockchain and Central and Eastern Europe Forum*. Ed. by C. Di Ciccio, R. Gabryelczyk, L. García-Bañuelos, T. Hernaus, R. Hull, M. Indihar Štemberger, A. Kő, M. Staples. Cham: Springer International Publishing, 2019, pp. 3–10. ISBN: 978-3-030-30429-4 (cit. on pp. 15, 16).

[Sza18] N. Szabo. “Smart Contracts : Building Blocks for Digital Markets”. In: 2018 (cit. on p. 15).

[Sza96] N. Szabo. “Smart contracts: building blocks for digital markets”. In: *EXTROPY: The Journal of Transhumanist Thought,(16)* 18.2 (1996), p. 28 (cit. on p. 15).

[TEK17] S. Tai, J. Eberhardt, M. Klems. “Not acid, not base, but salt”. In: *Proceedings of the 7th International Conference on Cloud Computing and Services Science*. 2017, pp. 755–764 (cit. on pp. 14, 22, 23).

[Woo16] G. Wood. “Polkadot: Vision for a heterogeneous multi-chain framework”. In: *White Paper* 21 (2016), pp. 2327–4662 (cit. on p. 27).

[WOY+19] S. Wang, L. Ouyang, Y. Yuan, X. Ni, X. Han, F.-Y. Wang. “Blockchain-enabled smart contracts: architecture, applications, and future trends”. In: *IEEE Transactions on Systems, Man, and Cybernetics: Systems* 49.11 (2019), pp. 2266–2277 (cit. on p. 15).

[YMRS19] D. Yaga, P. Mell, N. Roby, K. Scarfone. “Blockchain Technology Overview”. In: *CoRR* abs/1906.11078 (2019). arXiv: 1906.11078. URL: http://arxiv.org/abs/1906.11078 (cit. on p. 14).

All links were last followed on 2022-03-30

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

place, date, signature