INTEGRATION OF BLOCKCHAIN SMART CONTRACTS WITH ZEEBE PROCESS ENGINE AS TASK IMPLEMENTATIONS

Emre Özaras
S4102126

Supervisor: Prof. Dimka Karastoyanova

15 August 2022

Abstract

Blockchain applications and smart contracts provide novel means of communications and transactions between non-trusting parties, and they possess high potential for enhancing business processes. The use of blockchain smart contracts in business process management has been previously researched, but we believe the importance of automating workflows that contain the use of smart contracts together with other types of tasks, is overlooked. In this paper, we provide an approach for integrating the use of Ethereum smart contracts with the Zeebe process engine. We consider two use cases in order to illustrate the value added by smart contracts to business processes and explain our approach in detail.

Contents

1 Introduction

Workflow automation is the process of replacing jobs that are usually paper-based or manual with digital tools, usually using a single platform that allows extensive control over business processes.The main purpose of workflow automation is to provide enhanced control of resources and workforce allocation. Automating workflows allows companies to preserve their standards and provide visibility and accountability within their organizations. Different tools and applications are used for workflow automation. The focus on this paper is on Camunda Platform 8. Camunda Platform 8 makes use of a process engine called Zeebe for the execution of process instances. Users can model their workflows in BPMN, deploy, and execute them on Zeebe engine, all using Camunda Platform 8.

Emerging concepts in computer science are bringing new possibilities for businesses. A recent example is smart contracts, which -in very broad terms- are programs deployed to a blockchain network in order to automate transactions when a set of predetermined conditions are met. As blockchains and smart contracts are gaining more popularity and proving their worth, businesses are showing more incentives to incorporate these concepts into their business processes. By using smart contracts, businesses can remove intermediaries from transactions, reduce costs by replacing manual tasks, and store data safely. Therefore, integration of smart contracts with workflow automation tools would enable users to make their workflows more efficient and reap the benefits of these technologies. An important aspect to consider while working towards this integration is how to access smart contracts and their functionalities in order to use them within external applications. We are specifically interested in solutions that can be used in Zeebe client programs which implement tasks for processes in Camunda Platform 8.

In this paper, we provide a brief introduction to workflow automation, blockchain and smart contracts. We then go over related literature in Section 3. In Section 4, we explain technologies used in this project and in Section 5, we propose a solution for using smart contracts in business processes in Camunda Platform 8, go over two example business processes, and reflect upon our findings.

2 Background Information

In this section, the technological concepts related to the topic are explained. This section aims to provide the the basic information about different aspects of the project in order to prepare the reader for the Technologies Used Section.

2.1 Workflow Automation and Process Engines

Workflow automation can be defined as the design, deployment and automation of business processes where work tasks are automatically triggered and routed. It makes processes more efficient by replacing tasks made by humans with pieces of code that executes all or part of a process, usually through solutions that implement low-code approaches and adoption friendly user interfaces [9]. Workflow automation is used by businesses in order to visualize and manage their business processes better. A business process might be composed of multiple

services, possibly with some including manual tasks that can be automated, and visualizing the pipeline and tracking their status may be beneficial for firms to function more efficiently.

In workflow automation, processes are modeled using flowchart-like diagrams, often in the industry standard format BPMN [6]. In BPMN, users can use different components such as activities, events, and gateways to design their process models. After being designed, processes are deployed to process engines for execution. Process Engines are software applications that manage processing, storage, and distribution of data related to business processes and keep track of completion of process instances so that users can determine the status of processes [1].

2.2 Blockchain

A Blockchain can be explained as a distributed database that is shared between nodes of a network of computers. Blockchains store data in structures called blocks that are linked together by cryptographic functions [13]. Blockchains can technically be used for storing any sort of data. They are popularized after the invention of Bitcoin and cryptocurrencies and their most popular use is recording transactions. For that reason, it makes sense to take Bitcoin as an example while trying to understand blockchains better.

2.2.1 Bitcoin

Bitcoin is the first cryptocurrency and possibly the most popular implementation of a blockchain network. It was developed by pseudonymous person/group Satoshi Nakamoto with the aim of creating an en electronic currency network that is purely peer-to-peer. Bitcoin is defined as a chain of digital signatures by Satoshi Nakamoto in the Bitcoin whitepaper [16]. Owners transfer the coin to others by signing the hash of the previous transaction and the public key of the user they are going to transfer the money to, and adding these to the end of the coin. Payees then can verify the chain of ownership by verifying the signatures. A visualisation of this process can be seen in Figure 1.

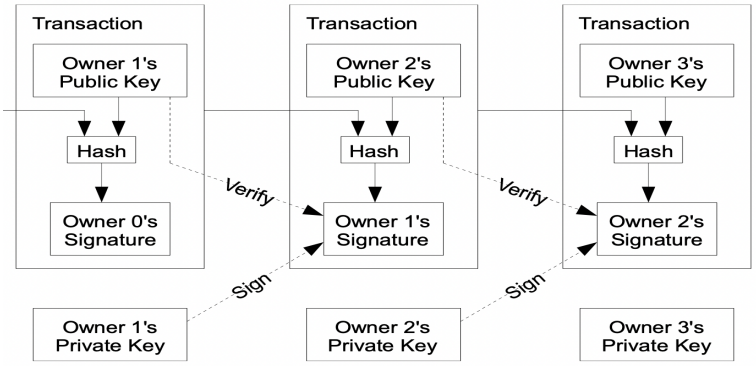

Figure 1: How transactions on Bitcoin are done [16].

In order to prevent owners to double-spend their money without using a trusted third-party, all transactions are publicly announced and then, nodes in the network agree on a single history of transactions. The solution proposed by Nakamoto is based on a timestamp server that takes hash of the block of items to be stamped and publishes it, which ensures

items existed at the time if they were in the hash [16]. Each timestamp includes the previous timestamp in its hash, and therefore forms a chain that is reinforced in each step. A visual representation of this process can be seen in Figure 2.

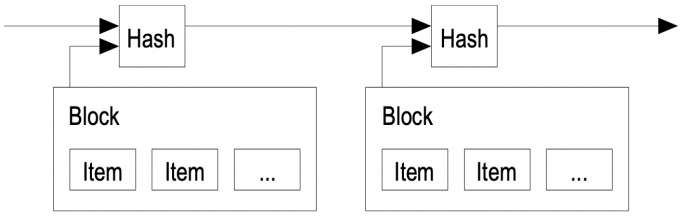

Figure 2: Timestamps in Bitcoin Network [16]

To implement a peer-to-peer distributed timestamp server, Bitcoin uses a proof-of-work system in which users spend CPU power to verify transactions and as long as more than half of the CPU power in the network are provided by honest nodes, the system works as intended[16]. Due to the way the network is organized, its practically impossible to tamper with the added blocks.

According to the Bitcoin whitepaper [16], the network functions as follows:

1. New transactions are broadcast to the nodes in the network.
2. Nodes collect new transactions into blocks.
3. Nodes work on finding a proof-of-work for their block.
4. When a node finds a proof-of-work, it broadcasts the block to all nodes.
5. Nodes accept the new block if all transactions in it are valid and not already spent.
6. Nodes express their acceptance of the block by working on creating the next block in the chain, using the hash of the accepted block as the previous hash.

Through the explained steps and features, Bitcoin replaced trust with computational power and cryptography and created a peer-to-peer currency that uses no intermediaries for transactions. As stated earlier in this section, Bitcoin draws an outline for the blockchain-based networks and provides an example use case for blockchain. However it is noteworthy that this does not mean all of its internal structure is present in other blockchain-based networks. Other cryptocurrencies may use different cryptographic functions and consensus algorithms while recording transactions. Also, recording transactions on transparent ledgers is only one use case of blockchain, and blockchains can be used for recording any sort of data.

2.3 Smart Contracts

Nick Szabo introduced smart contracts in 1994 and described them as computerized transaction protocols that execute terms of a contract [19]. Szabo described the objectives of a smart contract as follows [20]:

1. Observability
2. Verifiability
3. Privity
4. Enforceability

According to Kemmoe et al. [15] Bitcoin's read-append property and its scripting language showed that blockchain is a viable platform to implement smart contracts. When put on a blockchain, a smart contract can not be modified due to intamperability of blocks, it can be easily observed, verified, and privity also can be achieved depending on the access mode of the blockchain. They claim, although Bitcoins scripting language only allows for checking and validating transactions, Bitcoin can still be considered as an implementation of a smart contract on blockchain. However, Bitcoin's offerings regarding the creation and deployment of smart contracts on blockchain were very limited. This resulted in developers trying to create other blockchain based platforms. In 2015, Vitalik Buterin [3] took the essential concepts of Bitcoin and expanded them to create a blockchain platform called Ethereum which features a decentralized payment system with a coin called Ether, and a Turing-complete language for creating smart contracts on blockchain.

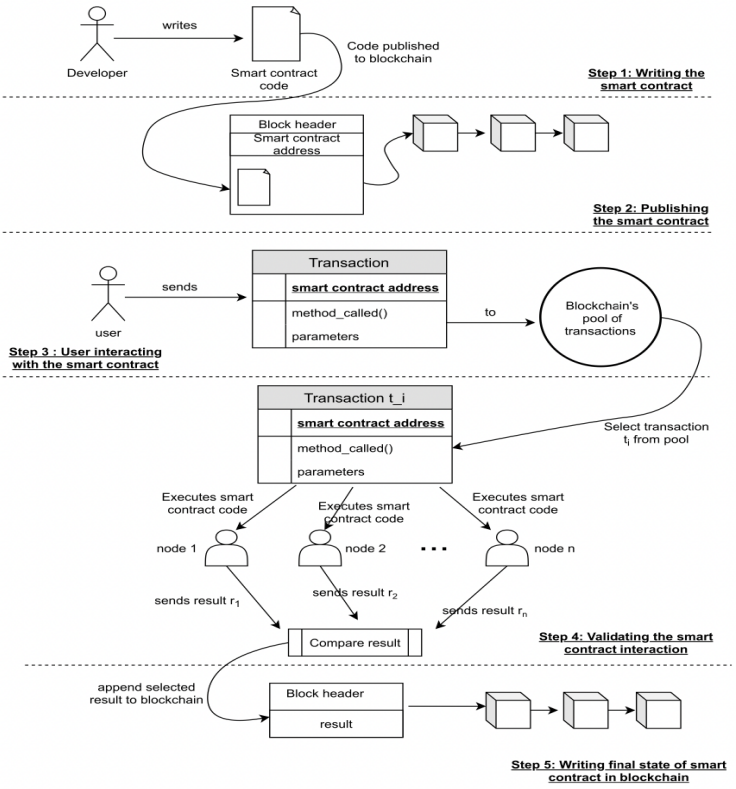

Figure 3: Overview of Smart Contracts [15]

Figure 3 provides a brief overview of how a smart contracts work. Step-by-step process regarding the working mechanism of smart contracts is as follows [15]:

1) Developers write contract logic in a language supported by the smart contract platform of choice. Then they obtain a bytecode by compiling the contract source code on a compiler that is often specified by the platform.

2) Developers publish the obtained bytecode to the blockchain platform and the contract is stored on a blockchain as a result. Depending on the platform, the contract may be read-only or modifiable after this step. If the platform restricts contracts to be read-only after being published, developers have to publish new contracts to provide updates and redirect users to the newer versions.

3) Users can now interact with the smart contract. Interaction protocols might differ from one platform to another. In the case of Ethereum, an address is returned to the user. Users can then use this address to interact with the smart contract by sending transactions. Transactions that are sent to the contract must contain the function of the contract users wish to invoke and its parameters. Transactions will be stored in the platforms transaction pool.

4) Sent transactions will be selected from the blockchain's pool of transactions to be executed. During their execution, intended functionality of the smart contract will be realized by a set of nodes. The involved nodes will then compare their results and keep one according to a consensus protocol.

5) Once a result is validated, it will be written into a block to be appended to the chain. Also, the smart contract's internal variables might be changed, if the invoked function is meant to change the contracts internal structure.

2.3.1 The Oracle Problem

Smart contracts execute automatically when specified conditions are met, and this may sound quite seamless at first. However, the mechanisms supporting this functionality is more complex then it seems. Sometimes, contracts need to get data from sources outside the blockchain environment. We can take the following pseudo smart contract as an example:

In this example, our contract needs to access weather data in order to determine whether ice cream sale occurs. However, reaching data that reside outside the blockchain is not easy for contracts. Ethereum, for example, does not allow smart contracts to query external websites as it would damage the deterministic nature of contracts since different nodes could receive different results for the same query [2]. A solution for this problem is providing an interface called an Oracle. Oracles are actually contracts themselves and their state can be changed by sending transactions[2]. Instead of querying an external service, a contract queries an oracle, and external services send transactions to oracles to update the data that they contain. Oracles are contracts themselves, so they can be queried without causing any consistency problems. Oracles often have on-chain parts that manage communication with other smart contracts, and off-chain parts that listen to events emitted and communicate with the intended APIs. A conceptual model of an oracle can be seen in Figure 4. In this model, a client smart contract makes a query by calling the newRequest() function of the oracle contract. Oracle contract, then, emits an event. Off-chain nodes of the oracle listen

Algorithm 1: Example Smart Contract

while Average Temperature is above 20 Degrees Today **do**
    Get *Money* from *Client* if given
    Get *Ice Cream* from *Seller* if given
    if *Ice Cream* and Money are equal to the agreed **then**
        Give *Money* to *Seller*
        Give *Ice Cream* to *Client*
        return
    end if
end while
Give *Money* back to *Client*
Give *Ice Cream* back to *Seller*
return

to events and reacts by querying the API. After receiving the response from the API, they update the data in oracle smart contract by calling the updateRequest() function. Then, query data is returned to the client contract that asked for it.

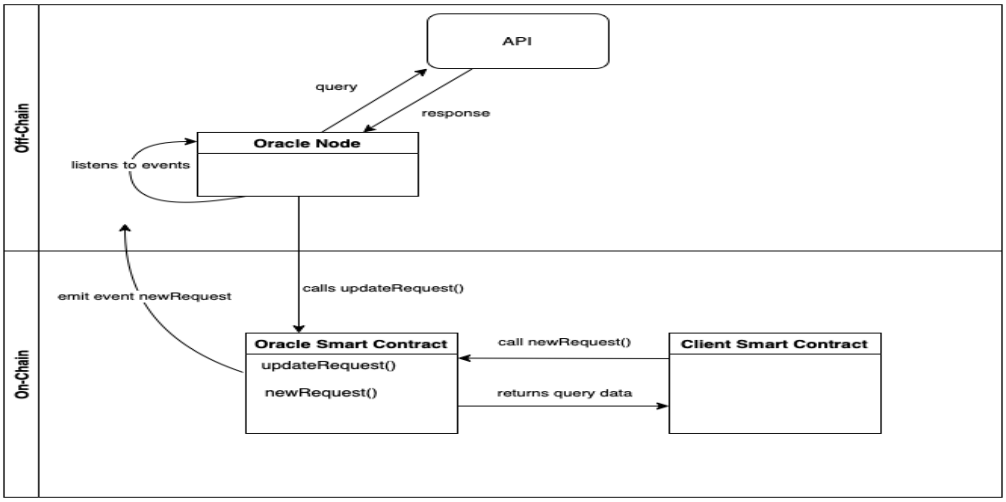

Figure 4: An Example Oracle, based on [8]

2.3.2 Growing Popularity

Since the introduction of Bitcoin and Ethereum, the interest for smart contracts have been growing. Companies started using them in different areas such as decentralized finance, gaming, and the legal industry. Considering the growing interest in Web3, which is characterized by decentralization and trustlessness [12], we can argue that their popularity will increase even further in the coming years. According to a ConsenSys press release, over 350.000 developers were using their blockchain development platform called Infura in 2021 [7].

3 Related Work

We searched for literature and applications that bring together workflow automation and blockchain/smart contracts in order to get a grasp of the current state of research regarding our topic. This section summarizes the literature and applications that we find important.

Data safety and decentralization provided by blockchain platforms have paved the way for many opportunities across different fields including business process management. Currently, most of the existing literature about the usage of blockchain smart contracts in the world of business process management is related to carrying some functionalities of the business process management systems on-chain. For example, Weber et al. [24] proposed to implement collaborative business processes on Ethereum. In the proposal, a BPMN choreography diagram, in which parties communicate through exchanging messages, is compiled into Solidity[1] code and messages between parties are sent as Ethereum transactions. Another application of blockchain in business process management is in runtime verification. Runtime verification can be defined as the evaluating whether a process execution meets the functional and non-functional objectives specified by process participants [18]. Prybila et al [18] recognized bitcoin blockchain's potential to provide a basis for implementing independent, decentralized, and undeniable runtime verification for business processes of the type choreography, in which control of the process is distributed among participants, and created a software prototype. Their system works on bitcoin with the usage of control tokens that store the state of the process execution. The party that possesses the control token is responsible for carrying out a part of the process, and then passes it to another party. The transaction chain of the token of a process instance contains undeniable proof regarding the progress of the execution for that process instance. This allows participants to claim rewards or penalties by proving their involvement in the process. In this system, data regarding the process execution are put into transactions that serve as handovers of control tokens. The data that is put into transactions include process id, id of the next task, a timestamp, process data hash, and receiver signature.

Writing smart contract code may be hard for developers that are not experienced with blockchain platforms. In such scenarios, model driven engineering (MDE) tools come to the rescue. Many tools that generate Solidity code from diagrams have been developed. Such translators lower the entry barriers into the world of smart contracts and have the potential to spread the use of smart contracts further in business processes. An example of this model driven tools can be found in Lorikeet [21]. However, Lorikeet itself is much more than just a translator, it allows users to design and execute blockchain applications in the domain of business processes. Lorikeet contains several different components within its system:

1) a BPMN to Solidity translator, which takes a BPMN diagram as an input and returns Solidity contracts as outputs.

2) a Blockchain registry generator that generates Solidity smart contracts that are to be used as registries.

[1]Solidity is a programming language used for developing smart contracts for Ethereum. See https://docs.soliditylang.org/en/v0.8.15/

3) a Blockchain trigger handles compilation, deployment and interaction with smart contracts.

This components of Lorikeet are designed as micro-services and deployed in individual Docker containers. Lorikeet, basically, generates a set of smart contracts from a business process and uses them to manage the business process. This makes Lorikeet a good tool for managing business processes on-chain.

A tool that is similar to Lorikeet is Caterpillar[17], which is a blockchain-based BPMN execution engine. Caterpillar's goal is to provide users with a system to build blockchain applications to ensure collaborative business processes, that are designed using the BPMN notation, are executed correctly. Caterpillar also translates BPMN models into smart contracts to achieve this goal. Different from other applications and literature, Caterpillar does not assume parties interact through message exchanges for coordination. It assumes that they use the blockchain as the only coordination mechanism instead. Caterpillar also maintains the state of each process instance and instances of their sub-processes on the blockchain.

The work that has been discussed in this section illustrates how much potential the intersection between fields of smart contract development and business process management carries. However,we believe, automating workflows that contain the use of smart contracts have not been explored enough. For that reason, demonstrating how to integrate smart contracts to workflows in Camunda might serve an important function as we identified this kind of integration as missing in the state-of-the-art.

4 Technologies Used

4.1 Camunda Platform 8

Camunda is a company that provides workflow automation services. Their latest product, Camunda Platform 8, allows users to design, automate and monitor business processes. In their website[4] they refer to it as "the universal process orchestrator". Camunda Platform 8 provides Camunda Modeler that allows users to design process diagrams using BPMN and DMN which are industry standard representations for business processes. They provide connectors that allow for easy communication with some popular enterprise applications.

4.1.1 Camunda Modeler and BPMN

Camunda allows users to design and connect their workflows using Camunda Modeler. This tool can be used online or can be downloaded and run locally. Camunda Modeler uses the industry-standard BPMN language to represent process diagrams. In BPMN, processes can be modeled using components such as events, tasks, variables and gateways. An example workflow model can be seen in Figure 5.

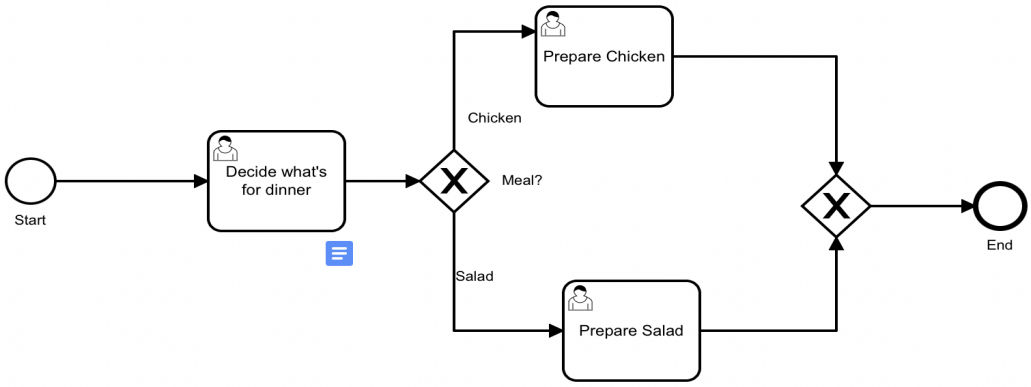

Figure 5: A BPMN Workflow designed in Camunda Modeler

Tasks are the center of focus for this project as we aim to implement them as smart contracts. There are different types of tasks within Camunda Platform 8. Figure 6 lists the types of tasks available in Camunda Platform 8.

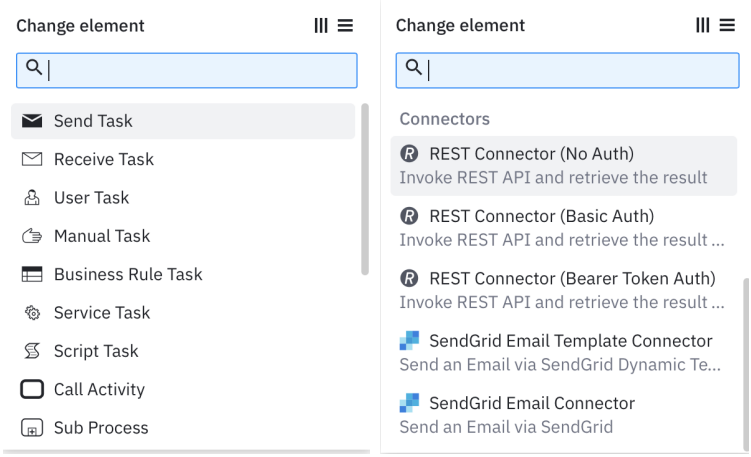

Figure 6: Tasks in Camunda Modeler

In this project, the focus will be on service tasks. Service task are tasks that are carried out through execution of a piece of code and communicates with the underlying system supporting the workflow automation through an API provided by Camunda. Service tasks are carried out through components called job workers. They will be explained in more detail in the following subsections.

4.1.2 Zeebe

Zeebe is a horizontally scalable, cloud-native, open-architecure BPMN workflow engine that is constituting the backbone of Camunda Platform 8. Zeebe's architecture can be seen in Figure 7:

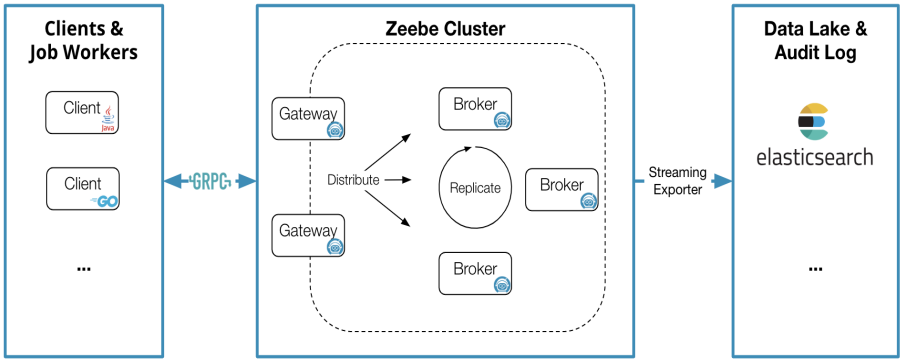

Figure 7: Zeebe's Architecture [5]

Camunda provides documentation regarding Zeebe's architecture [5]. According to this documentation, Zeebe's architecture is composed of three layers: Clients and Job Workers, Zeebe Cluster, and Data Lake Audit Log. Client programs can be implemented in any language that is supported by the Zeebe API and send commands to Zeebe to: (1) deploy processes, (2) carry out business logic, (3) handle operational issues. Clients are libraries that can be embedded in the code of services in order to connect to Zeebe. Job workers are a specific type of Zeebe client programs that use the client API to start a job, and either complete it or fail it in the end of the process. Zeebe Cluster is composed of two parts: Gateway and Brokers. Gateway is the access point to the cluster and requests sent to it is forwarded to Brokers. Brokers are the key to the operations of Zeebe: a Zeebe broker is the distributed process engine that keeps the state of active process instances. The distributed structure of the system enables deployments to scale horizontally. Brokers form a peer-to-peer network and duties of a busy broker will be transparently reassigned to another broker. On the SaaS usage of Camunda Platform 8, users work exclusively with clients, and other components are only relevant in local or private cloud deployments.

4.1.3 Job Workers and Service Tasks

Job workers are services that perform specific tasks in processes. Each time a service task needs to be done, it uses a corresponding job worker [4]. A job contains a type, custom headers, key and variable(s). Zeebe officially provides Java and Go clients, however it is possible to code in other languages as well [4]. Job workers are configured with credentials in order to connect to clusters. Then, they will be invoked when a workflow requires a task that matches their type. An example Zeebe job worker can be seeen in Figure 8.

```java
@Component
public class EmailWorker {

 @ZeebeWorker(type = "email", autoComplete = true)
 public void sendEmail(final ActivatedJob job) {
   final String message_content = (String) job.getVariablesAsMap().get("message_content");
   LOG.info("Sending email with message content: {}", message_content);
 }
}
```

Figure 8: A Zeebe worker written in Java [4]

4.2 Ethereum

We chose to use Ethereum as the smart contract platform for this project because:

- It is the first smart contract platform, and has a team with great dedication for smart contracts.
- The coin it supports (ETH) has a large market cap.
- Many developer tools and SDK'S are available for Ethereum development.

In her article [14], Kasireddy describes the structure of the Ethereum network and how it works. The Ethereum blockchain can be defined as a transaction-based state machine . Etherum state machine begins with a so-called "Genesis State", this is when no transactions have been made on the network. As transactions go through, machine transitions into some "final state". The "final state" in any given time represents the state of Ethereum at that time. A visualisation of Ethereum state transitions can be seen in Figure 9.

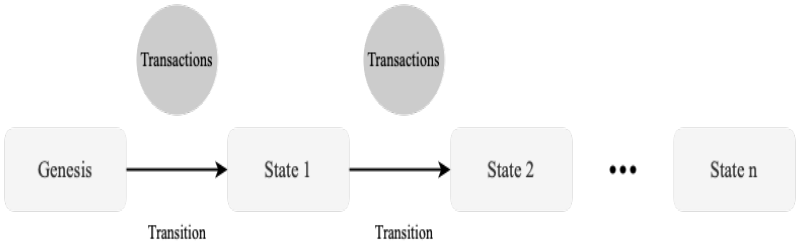

Figure 9: Ethereum State Transitions, based on [14]

For a transaction to cause a transition, that transaction should be recognized as valid. This is achieved through a process called mining. Mining requires a group of nodes to use computational resources to create a block of valid transactions that is to be appended to the chain. In order to append a new block to the chain, a miner must prove it faster than any other competitors by providing a computational proof called "proof-of-work". A miner that validates a new block will be given a specific amount of value (in Ethers for Ethereum).

4.2.1 Accounts

Global state of Ethereum is constituted by many objects that interact with each other through message passing [14]. These objects are called accounts. Each account has a state and an address. There are two types of accounts:

1) Externally owned accounts that are controlled through private keys and are not associated with a piece of code.
2) Contract accounts which are controlled by their contract code.

External accounts can send transactions to other external accounts or contract accounts by signing transactions with their private keys [14]. Contract accounts, on the other hand, cannot be used to send transactions on their own unless they are triggered by the conditions written in their contract code.

4.2.2 Gas Fees

Gas fees are a very popular concept related to Ethereum network. Ethereum network charges a fee for every transaction due to computational costs [14]. This fee is called "Gas" . In every transaction, the sender sets a gas price and a gas limit, and their product represents the maximum value they are willing to pay as a transaction fee. How gas fees work in Ethereum can be seen in Figure 10.

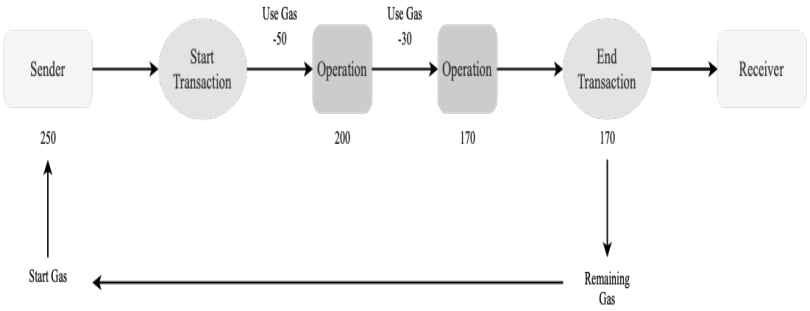

Figure 10: Gas in Ethereum, based on [14]

If the sender fails the provide the necessary amount of gas for a transaction to go through, then the transaction is considered invalid and gets reverted [14].

4.2.3 Transactions and Messages

A transaction in Ethereum network can be explained as a cryptographically signed instruction that is initiated by an externally owned account [14]. There are two types of transactions:

1) Message calls

2) Contract creations

Contracts that exist within the global scope of the Ethereum's state can communicate with other contracts within the same scope through "messages" or "internal transactions". How internal transactions happen can be seen in Figure 11.

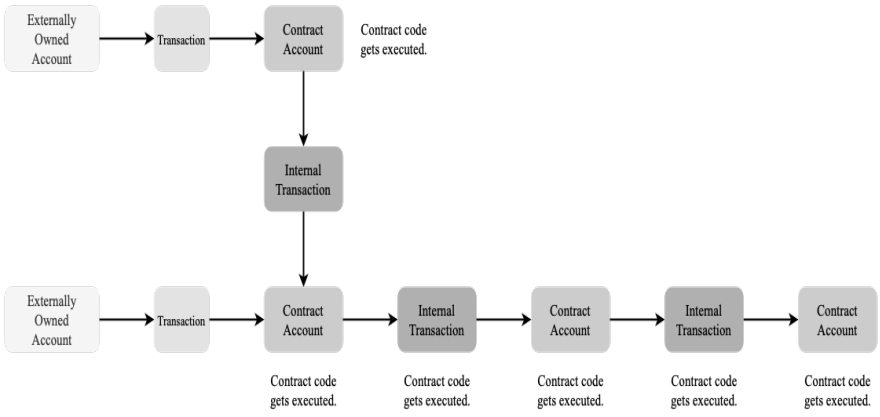

Figure 11: Internal Transactions in Ethereum, based on [14]

Internal transactions do not include a gas limit, therefore the gas limit set by the sender of the first transaction must be high enough to cover all sub-executions and contract-to-contract messages as well. Ethereum generates a receipt for every transaction and this receipt contains items such as:

- Block number
- Block hash
- Transaction hash
- Gas used in the transaction

4.2.4 Smart Contracts in Ethereum

Smart contracts in Ethereum are usually written in programming languages called Solidity and Vyper and structured similarly to Java classes. They may contain some data fields together with constructors, getter and setter functions. Ethereum requires smart contracts to be in bytecode in order to be executed by the Ethereuum virtual machine [10]. In other words, contracts must be first compiled before they are deployed to the Ethereum blockchain. After compiling the contract and acquiring the bytecode, users can deploy contracts to Ethereum by sending transactions [11]. In order to deploy a contract successfully, the following are required:

- Contract's bytecode
- Enough ETH to cover gas fees
- A deployment script or plugin
- Access to an Ethereum node

4.3 Infura

Normally, developing for blockchain platforms require developers to run a node of the blockchain platform they are developing for. However, this is often time-consuming and it consumes processing power. Infura provides infrastructure for Ethereum projects and simplifies blockchain development. It supports the Ethereum mainnet, together with testnets such as Rinkeby, Goerli, Kovan, and Ropsten. This is ideal for this project since the operations are performed on Rinkeby testnet. Also, not having to run a node locally simplifies writing and managing code that is to be used within a workflow.

4.4 Web3j

Web3j is a highly modular and reactive Java and Android library for developing smart contracts and integrating with Ethereum nodes [23].Web3j allows developers to communicate with Ethereum network in their Java projects without having to write code for integration. How Web3j manages the communication with Ethereum network can be seen in Figure 12. Web3j features:

- Full implementation of Ethereum's JSON-RPC client API[2] over HTTP and IPC
- Ethereum Wallet support
- Auto-generation of Java smart contract wrappers in order to create, deploy, transact with and invoke smart contracts from native Java code
- Support for Infura clients
- Support for ERC20[3] and ERC721[4] token standards

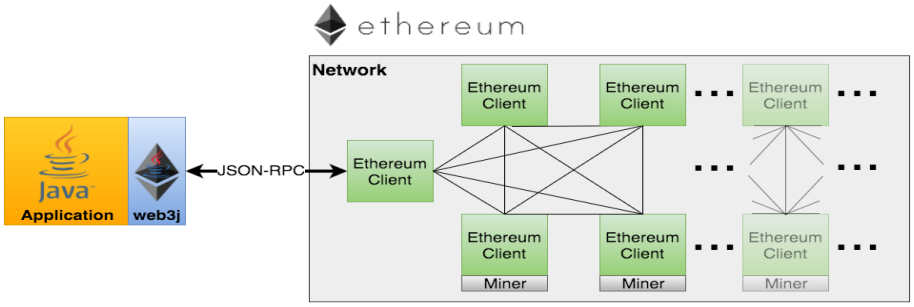

Figure 12: Web3j and Ethereum Network[23]

Web3j is an ideal library to use since it provides connect to Ethereum through Java code and Java is an officially supported language for Zeebe Clients. Moreover, Web3j supports Infura, the node provider we chose to work with.

5 Implementation

5.1 Solution Architecture

In our project, we used Zeebe clients that include the business logic written in Java. Zeebe clients communicate with the Zeebe Gateway to interact with processes deployed on the Zeebe Cluster. Interaction occurs via a gRPC API[5] and is simplified by containing the Zeebe clients within a Spring Boot application. The interaction with smart contracts occur via JSON-RPC requests sent to the Ethereum client provided by Infura. JSON-RPC requests are wrapped by Web3j library and the logic behind them, therefore, is written in Java. Architecture of the proposed solution can be seen in Figure 13.

[2]See https://ethereum.org/en/developers/docs/apis/json-rpc/
[3]See https://ethereum.org/en/developers/docs/standards/tokens/erc-20/
[4]See https://ethereum.org/en/developers/docs/standards/tokens/erc-721/
[5]See https://grpc.io/

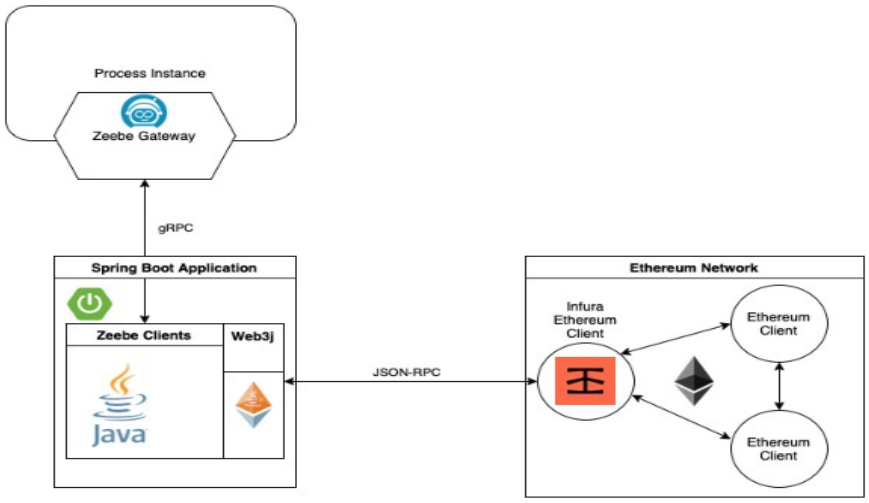

Figure 13: Architecture of proposed solution

In this solution, the interfaces through which the communication between the components occur were actually made by our technology selections. Zeebe uses gRPC as means of communicating with workers, and every Ethereum client must satisfy the JSON-RPC specifications as prescribed by Ethereum network. Additional components may put in between to make client code communicate indirectly if it is desirable in any situation.

5.2 Setting-up

We started writing our client programs using Spring Boot [6]. Using Spring Boot is not necessary for writing Zeebe clients, however, it simplifies writing code and makes the application easier to read. We then needed to connect our application to our Camunda cluster by putting in our cluster ID, client ID and client secret in the yaml file. Then we installed web3j dependencies in our project.

5.3 Connecting to Infura

We can send requests to our Infura node by setting up an Infura project, selecting a network to connect and then setting up a HTTPService using the endpoint given by Infura. It looks like this in Java:

```
1     Web3j web3j = Web3j.build(new HttpService("https://rinkeby.infura.io/
          v3/USER-SECRET"));
```

Listing 1: Connecting to Infura

5.4 Transactions

Implementing simple transactions were quite straightforward: passing wallet addresses as parameters in the process diagram and then using the methods exposed by Web3j in job

[6] See https://spring.io/projects/spring-boot

workers were adequate. After a transaction is made, transaction receipts or transaction hashes can be used in another task to check the validity and details of a transaction.

5.5 Deploying Smart Contracts

Deploying smart contracts were a little more complicated. In order to deploy instances of a smart contract to Ethereum using Web3j methods, we first need to generate a Java class that serves as a wrapper to the contract. In order to achieve this wrapper class, we first need to compile the contract written in Solidity. This can be achieved with the following command after installing Solidity [22]:

```
$ solc <contract>.sol --bin --abi --optimize -o <output-dir>/
```

Compilation should result in creation of two files: binary code and abi file. We can use those files to generate the Java wrapper class using the following command from Web3J command line interface[22]:

```
web3j generate solidity -b /path/to/ <smart-contract>.bin -a /path/to/<smart

-contract>.abi -o  /path/to/src/main/java -p com.your.organisation.name
```

Our Java wrapper class is then generated and instances of the contract can be deployed to Etherum using the parameters that have been passed from the process diagram through Web3j methods.

5.6 Interacting with Smart Contracts

Interacting with smart contracts can be done in two ways:

- Instantiating an object of the java wrapper class of the contract by using the load function with the given address and calling functions of the wrapper class:

```
1 YourSmartContract contract = YourSmartContract.load("0x<address>|<
    ensName>",
2 <web3j>, <credentials>, GAS_PRICE, GAS_LIMIT);
3
4 TransactionReceipt transactionReceipt = contract.someMethod(
5         <param1>,
6          ...).send();
```

Listing 2: Interaction Method 1

- Encoding functions using web3j's function encoder and sending them as a part of the transaction to the contract address:

```
1 Function function = new Function("functionName", Collections.emptyList
    (),
2 Collections.emptyList());
3
4 String txData = FunctionEncoder.encode(function);
```

```
5  TransactionManager txManager = new RawTransactionManager(web3j,
       credentials);
6  String txHash = txManager.sendTransaction(
7                  DefaultGasProvider.GAS_PRICE,
8                  DefaultGasProvider.GAS_LIMIT,
9                  contractAddress,
10                 txData,
11                 BigInteger.valueOf(value)).getTransactionHash();
```

Listing 3: Interaction Method 2

In this project we used both methods. We used the first one when reading values from a contract, and the second one when writing to the contract. It is easier in the first method to read variables because return types of functions are already known from Java wrapper class of the contract and extra decoding is not necessary. Second method provides more control over the amount of money transferred with the function call.

5.7 Evaluation

In this section we will go through two example workflows in order to provide a proof of concept for our proposed solution.

5.7.1 Example: Sales Workflow

One of the most popular use cases of cryptocurrencies, and perhaps the reason behind their existence, is allowing transactions to go through without the involvement of third parties. Therefore, in this project we decided to utilize smart contracts to carry out safe payments in an online sale scenario. The idea is to utilize an Ethereum smart contract as a payment channel through functions that can either be called by buyer or the seller to complete the transaction. The process will go as follows:

1) Buyer will place an order including their Ethereum wallet address (analogous to their credit card number)

2) Seller will deploy a SafePayment contract on Ethereum using their own wallet address and the wallet address given by the buyer.

3) Seller will send the contract information to the buyer.

4) Buyer will deposit money to the contract using a specific function on the contract, altering the state of the contract.

5) Seller will check the status of the contract to see whether enough money has been put into the contract or not. If so, they will ship the ordered goods. If Buyer does not deposit enough money into the contract or the ordered goods are not available anymore, seller can cancel the sale and money deposited by the buyer will be returned.

6) Buyer will receive the goods and verify delivery by using a function on the contract.

7) Money then will be transferred from contract to Seller's wallet address.

It is also possible to add expiration dates to contracts, so that contract becomes invalid after a certain time and order gets cancelled if the buyer does not send the money in time or seller does not deliver goods in time. This feature is not implemented in this example in order to keep the code as simple as possible. Even though this process looks rather meticulous at the first glance, it includes less stakeholders then a payment by credit card, and does not necessarily include more complex communication overall. Moreover, the gas fees that are used throughout the processes are likely to be much lower than the fees that would be paid to third party payment service providers. A simplified workflow diagram is can be seen in Figure 14.

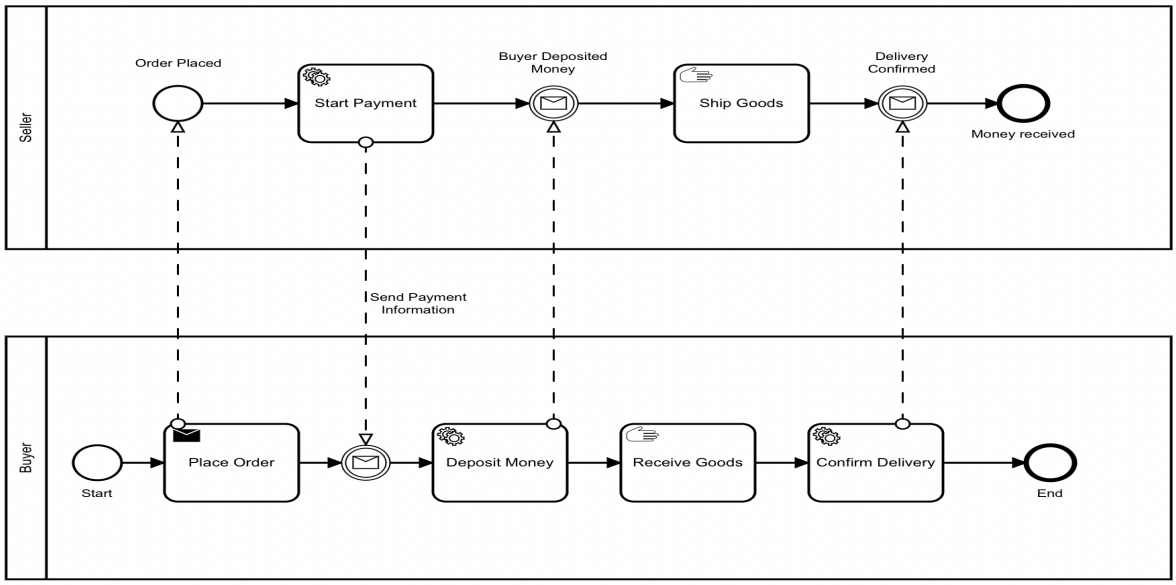

Figure 14: An Example Workflow

The Solidity code used for the contract are as follows:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.13;

contract SafePayment {
    address payable public owner;
    address payable public buyer;
    uint256 price;
    uint256 totalBalance;
    bool bought;

    constructor(address payable _buyer, uint256 _price) {

        owner = payable(msg.sender);
        buyer = _buyer;
        bought = false;
```

```
16          price = _price;
17
18      }
19      function deposit() public payable {
20          uint256 amount = msg.value;
21          totalBalance+= amount;
22          if(totalBalance>= price){
23              bought = true;
24          }
25
26      }
27
28      function isBought() public view returns (bool) {
29          return (bought);
30      }
31
32      function verify() public {
33          require(msg.sender== buyer, "Only buyer can verify delivery!");
34          uint amount = address(this).balance;
35
36          // Send all Ether to owner
37          (bool success, ) = owner.call{value: amount}("");
38          require(success, "Failed to send Ether");
39      }
40
41      function cancel() public {
42          require(msg.sender == owner, "Only owner can cancel purchase!");
43          uint amount = address(this).balance;
44          (bool success, ) = buyer.call{value: amount}("");
45          require(success, "Failed to send Ether");
46
47      }
48  }
```

Listing 4: SafePayment Contract Code

As can be inferred from the Solidity code given above, we used the addresses passed to the constructor in require clauses to make functions customer only or seller only. This ensures that parties do not interact with the contract on behalf of each other for malicious purposes. Buyers use the deposit function to deposit money into the contract, if enough money is deposited, the value of a corresponding Boolean variable will be assigned to true. Seller can then check the value of this variable to determine whether enough money has been deposited to the contract.
We then compiled this code into .bin and .abi files and proceeded to generate a Java wrapper class using the Web3j command line interface. Then we were able to deploy instances of SafePayment contracts through Zeebe job workers.

In order to test our solution, we designed and deployed a workflow diagram that depicts a safe payment scenario. Transactions in Ethereum require some time to be validated, so we decorated the diagram with timer events that occur after each transaction. Timer events help maintaining the order of transactions. The resulting diagram can be seen in Figure 15.

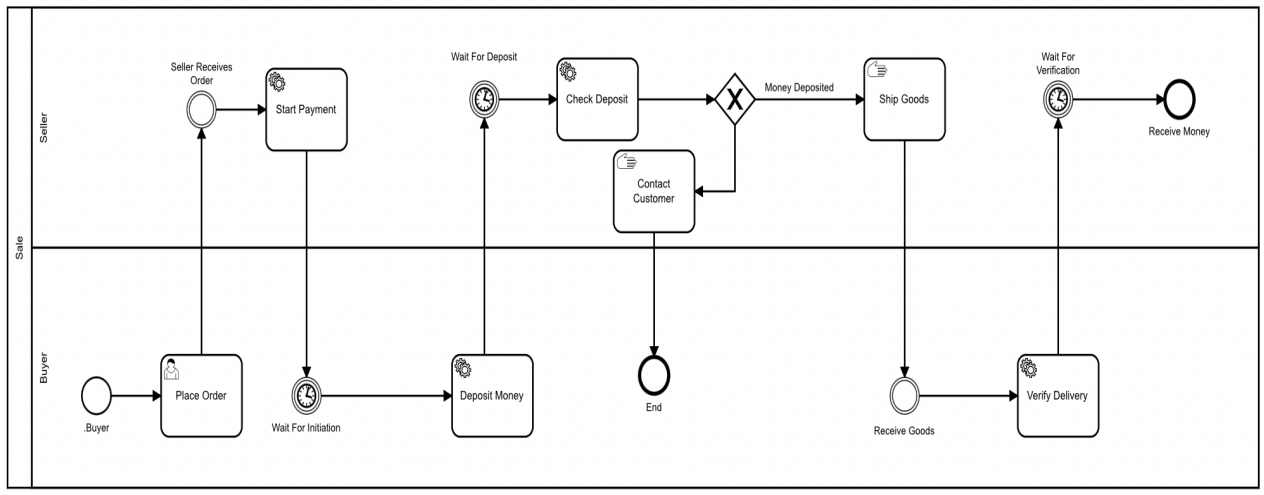

Figure 15: Safe Payment Workflow with Timers

We continued by coding job workers to implement the necessary service tasks. Our first worker corresponds to the service task "Start Payment". It retrieves the Ethereum wallet address of the customer and the price of the item to be sold from the process variables and uses them in the constructor of the contract to be deployed and initiated. This contract that would serve as the payment channel for the sale process. Our second worker is for depositing money. It gets the Ethereum address of the contract from the process and sends a transaction using the deposit function and adds the hash of the transaction into process variables. Our next worker checks the value of a Boolean variable to determine if enough money has been put into the contract. Our final worker in this scenario calls the verify function in our smart contract and triggers the transfer of funds from the contract to seller's wallet. Implementations of these workers can be found in the appendix.

After writing and connecting job workers, we deployed the diagram into a cluster and run an instance of it. As can be seen in Figure 16, the instance was completed successfully.

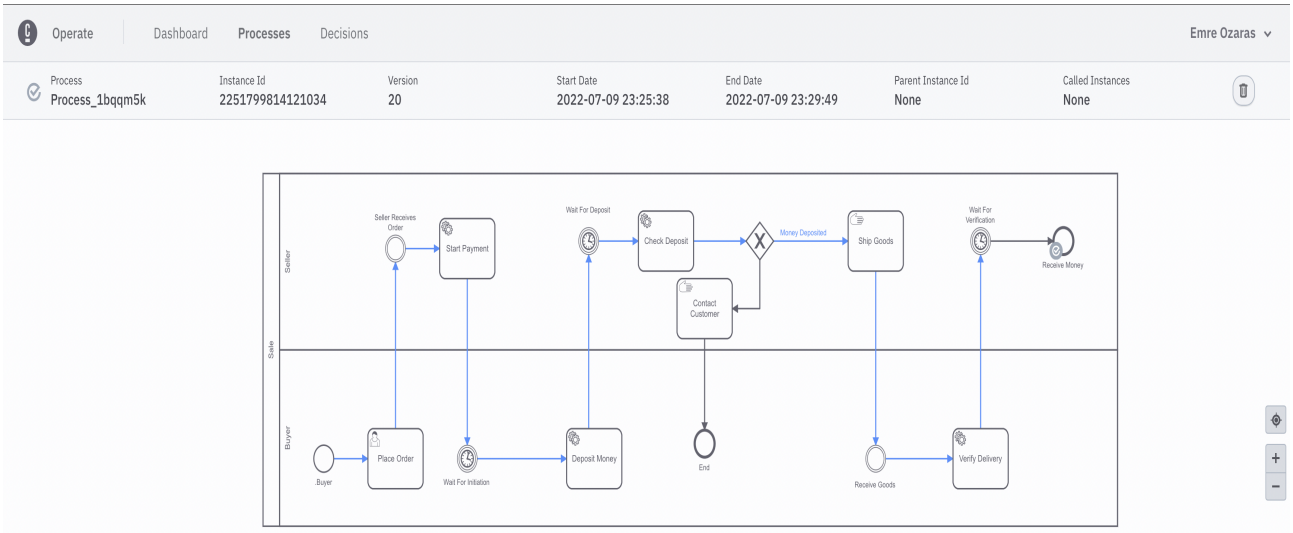

Figure 16: Safe Payment Workflow Instance

In each job worker, we put the included Ethereum addresses and transaction hashes into process variables in order to double check our implementation using Rinkeby Etherscan, an online tool that monitors transactions on Rinkeby Ethereum Testnet . We have observed that the expected transactions can indeed be seen on Rinkeby Etherscan, which indicates interaction with Ethereum network worked successfully in our solution. Some of the variables put into the process can be seen in Figure 17 and Rinkeby Etherscan page of the contract mentioned in the variables can be seen in Figure 18.

Variables

Name	Value
contractAddress	"0xd872b5668c5a5c01f3997af8d7dfa860351eeb7f"
creationHash	"0xead8302581fd0d754a8d2034f5a6f3f5ae7a7aac63f5c3d53ccb63f17d58762c"
paymentStatus	"Successful"
price	500000000
transactionHash	"0xc26b2e243da56a7e8900f940ae3422463a6434e13ab17207847711a38f4b6af9"
verificationHash	"0x0927ee6a62f94af71eb28d97ccffeb8622d63cc6b6ad55ded9d853b755798fb2"

Figure 17: Some Variables in SafePayment Workflow Instance

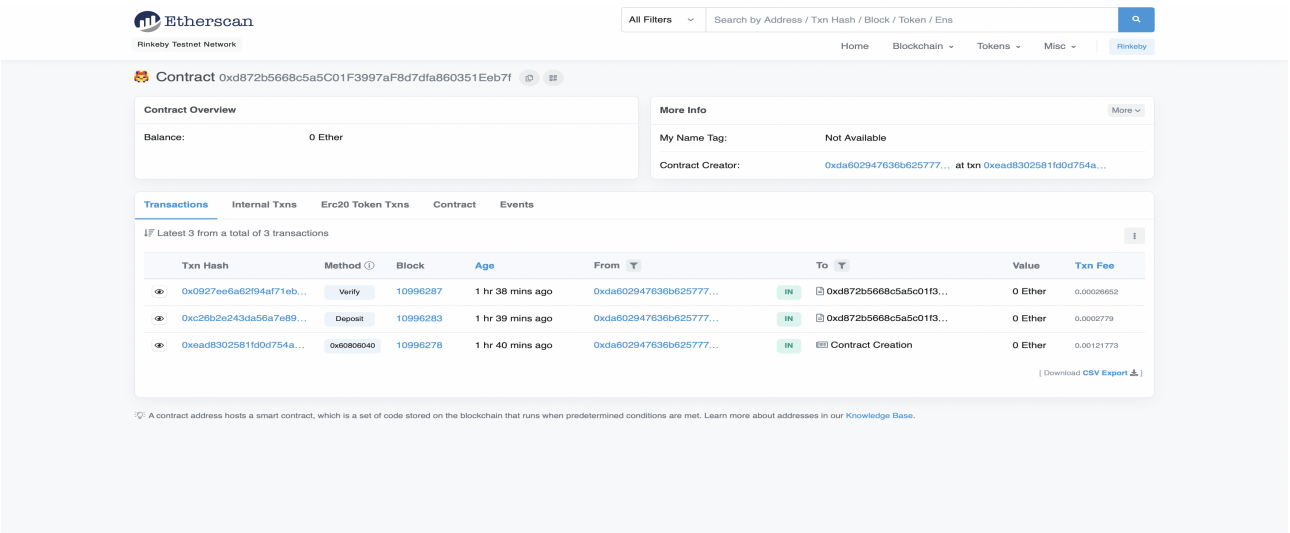

Figure 18: Contract on Rinkeby Etherscan

5.7.2 Example: Donations Workflow

Another use case of smart contracts may be for collecting donations. In our second example we will consider the story of a charitable organizations that wants to receive donations

through a smart contract for one of their campaigns. Each donation will be logged and the person makes the highest donation will be invited to the gala of the campaign. The related process can be modeled as follows:

1) Charitable organization deploys a smart contract and shares its address with public.

2) People make donations by calling a function on the smart contract, which takes an email address as a parameter.

3) When the deadline for donations come, charitable organization claims the money from the contract.

4) Charitable organization retrieves the email address of the highest donor and sends an invitation.

It is noteworthy that the deadline of a campaign can be managed through either smart contract itself or the process model. We decided to do the second one in order to keep smart contract simpler. The process diagram created for this workflow can be seen in Figure 19.

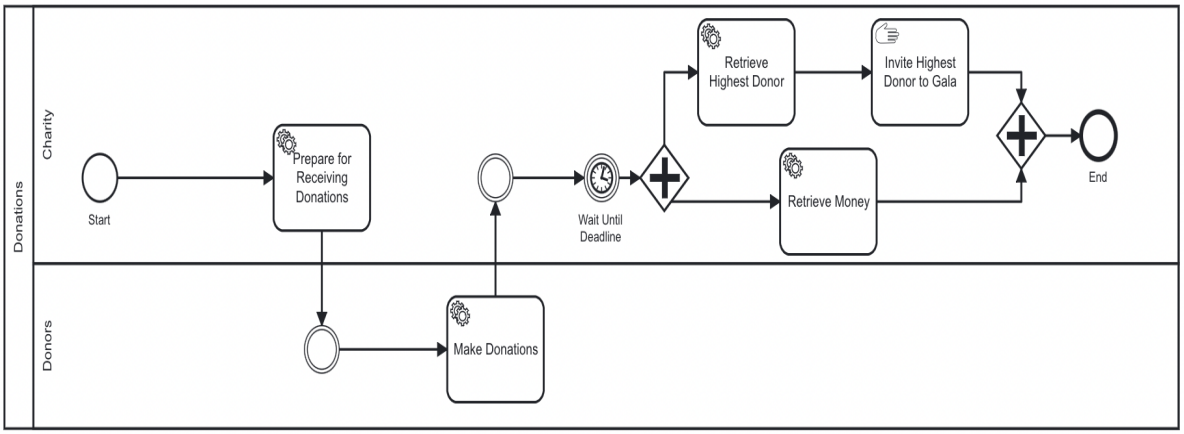

Figure 19: Donations Process Diagram

In this process diagram, the task "Prepare for Receiving Donations" is connected to a job worker that deploys a smart contract which will be explained later on. The task "Make Donations" is connected to a worker that takes the contract address and makes donations with different email addresses for testing purposes. Generation of donations can be seen in Figure 20. If our process is executed as intended, the email retrieved in the end should be "highestdonor@gmail.com", because highest amount is sent together with this email. In the implementation, we used sleep function to ensure there is enough time after a transaction so that it can be verified before the next one is sent. Then, a timer event is introduced to make the process wait until the campaign is over. After the deadline, money in the contract is transferred to organization's own account and the highest donor's email is retrieved. Then another job worker or a connector can be configured to send an email automatically, but since this project's focus is on smart contracts, we did not implement this functionality and left the invitation task as a manual task. The Solidity code for the smart contract that is used in the process described can be seen in Listing 5.

Figure 20: Generating Donations in a Job Worker

```
    pragma solidity ^0.8.7;
contract Donation {
    address payable owner;
    uint256 totalDonationsAmount;
    uint256 highestDonation;
    address payable highestDonor;
    string HighestDonorEmail;

    constructor(address payable _owner)  {
        owner = _owner;
        totalDonationsAmount = 0;
        highestDonation = 0;
        HighestDonorEmail = "";
    }

    event DonationMade(
        address  _donor,
        uint256 _value,
        uint256 _timestamp
    );

    function deposit(string memory email) public payable {
        uint256 donationAmount = msg.value;

        emit DonationMade(msg.sender, donationAmount, block.timestamp);

        totalDonationsAmount += donationAmount;

        if (donationAmount > highestDonation) {
            highestDonation = donationAmount;
            highestDonor = payable(msg.sender);
            HighestDonorEmail = email;
        }
    }

```

```
36      function claim() public payable {
37          require(msg.sender == owner, "Only owner can call this
                functionality!");
38          (bool success, ) = owner.call{value: totalDonationsAmount}("");
39          require(success, "Failed to send Ether");
40      }
41
42      function getHighestDonation() public view returns (string memory)  {
43          return (HighestDonorEmail);
44      }
45
46      function getTotalDonationsAmount() public view returns (uint256) {
47          return totalDonationsAmount;
48      }
49
50      function destroy() public {
51          require(msg.sender == owner, "Only owner can call this function!")
                ;
52          selfdestruct(owner);
53      }
54  }
```

Listing 5: Donation Smart Contract Code

In the Donation contract, constructor takes owner's address as a parameter. This is used in claim and destroy functions since these functions should only be called by the owner of this contract. Donors can make donations by calling the deposit function with some ether and their email address put in as a parameter. A DonationMade event will then be triggered and their address, the amount they have donated and the time of donation will be logged to Ethereum blockchain. If their donation is higher than the existing highest donor at the time, contract will be updated and they will be recognized as the highest donor. Money in the contract can be taken by the owner organization by calling the claim function.

We have tested our job workers by deploying the diagram described before in this section and executing an instance of it. As can be seen from Figure 21, our test was successful. We made each job worker put some variables such as addresses and hashes into the process for testing purposes. We then checked the transactions by searching the hashes on Rinkeby Etherscan. We have observed that the expected transactions can indeed be seen on Rinkeby Etherscan.

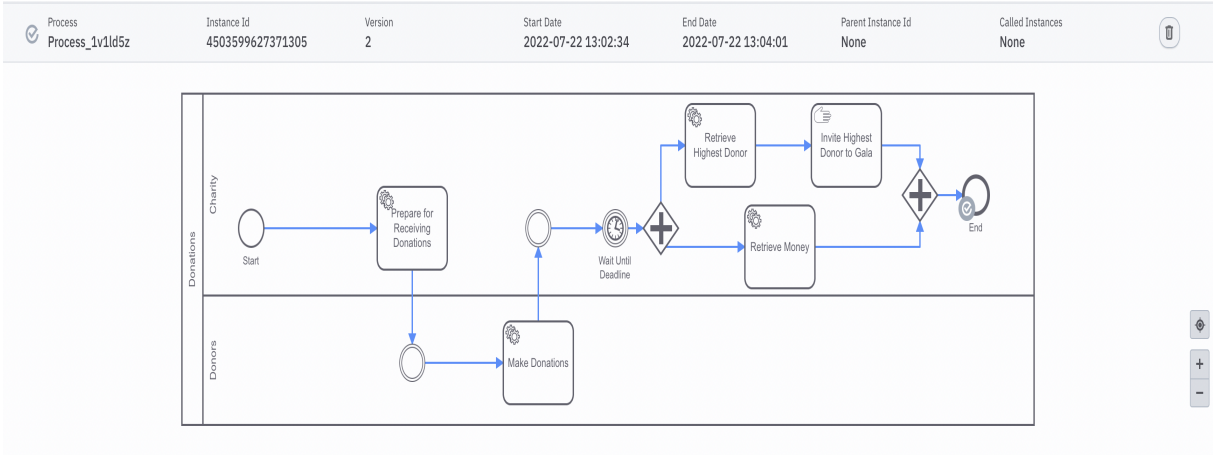

Figure 21: Instance of Donation Process

In Figure 22, you can see some of the variables put into the process by the job workers. The variable highestDonorEmail has the value "highestdonor@gmail.com" which shows that the smart contract code and the service workers worked as intended in updating and retrieving the email address of the highest donor.

Variables

Name	Value
Hash of 2	"0xc70b5423be020cb7607442a38d1ec9df639fa4a8d28bbf81c3b562d4e48c5c87"
Hash of 3	"0x7205b22196009cf41f2e440833e48086de765bc14b6565512d90f39d91270fcd"
contractAddress	"0x9b352e846229d71e33b1d5cd0ad2f498002c43f1"
creationHash	"0x1e98f3fb581ea5e7c881304d4ed81bf4c03065f86aebf2558b6aa9de62c52059"
highestDonorEmail	"highestdonor@gmail.com"

Figure 22: Variables of a Donation Process

When we go to the contract's page on Rinkeby Etherscan, we can see the transactions done on it. This can be seen in Figure 23.

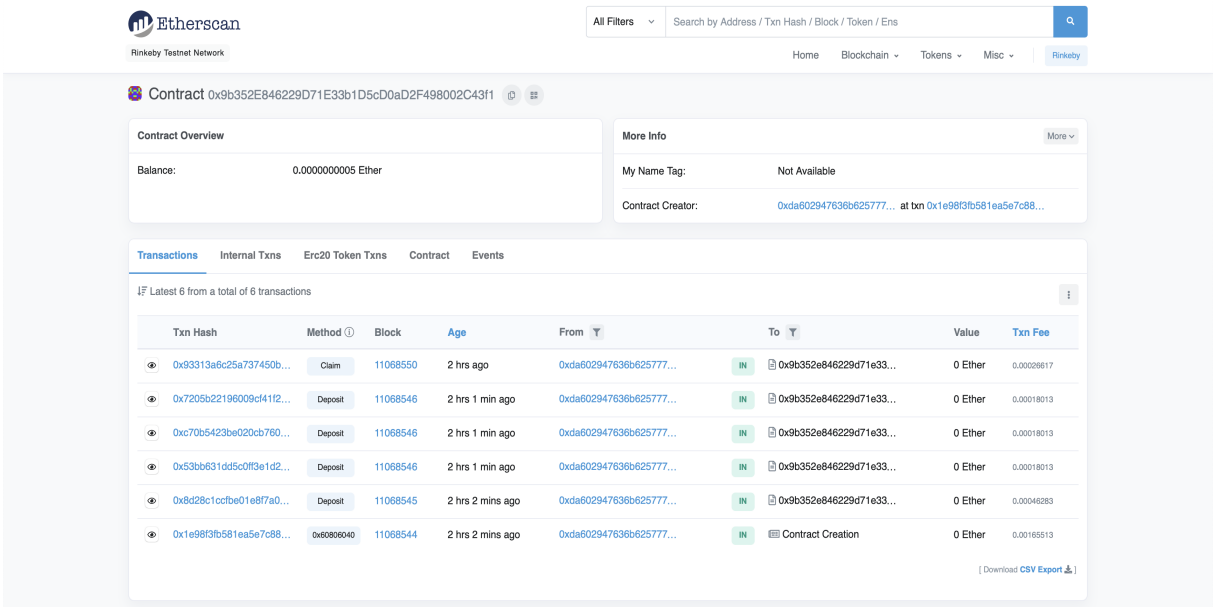

Figure 23: Donation Contract on Rinkeby Etherscan

To check if the claim function worked well and money has been transferred to owners account, we go to internal transactions tab and click on the hash of the transaction. In our case, money in the contract has indeed been transferred to the owner as can be seen from Figure 24.

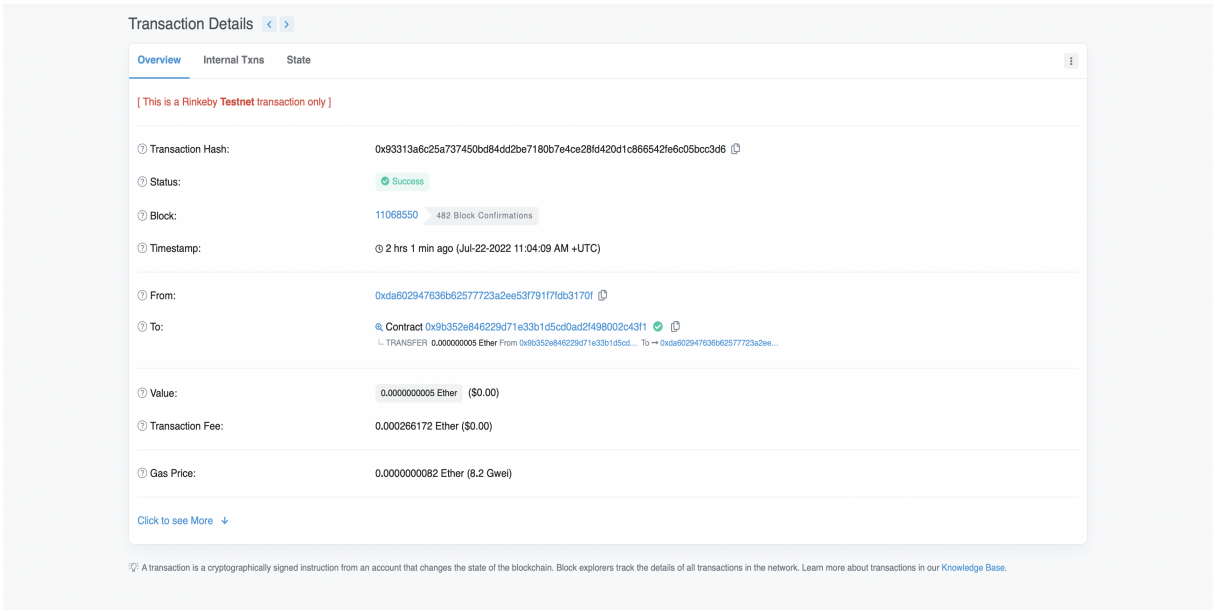

Figure 24: Claim Money Transaction on Rinkeby Etherscan

5.8 Problems Encountered

One of the problems encountered during this project was the waiting time for transactions. Due to the way Ethereum works, transactions are not instantly verified and added to blocks. This creates problems both in monitoring transactions and implementing business logic. For

example, if we have a workflow that first deploys a contract and then sends money to it, we need to wait after the deployment call to ensure there is a contract before we send Ethers to it. Otherwise, the transaction might be sent to an address which is not connected to any contract yet. This may cause the money that has been sent to be irretrievably lost. We have used two different solutions for this issue when we have encountered it. The first time we encountered this problem was when depositing Ethers after deployment of a contract in example sales workflow. We have used one minute long timer events in the process diagram after each service task in the process diagram and this solved the problem for us. We then ran into the same problem in example donations workflow when we are generating donations. This time, because we make all donations in the same worker, instead of using timer events in the process model, we made the thread sleep in the Java application for a while after each transaction. This also seemed to solve the issue for us, at least for this specific case. However, this asynchronous nature of Ethereum can be problematic for inexperienced programmers.

Another issue was managing gas fees and gas limits. Web3j has built-in default values for gas fees and gas limits that can be accessed through default gas provider. However, in transactions that may trigger other internal transactions, gas limit and gas prices may need to be increased for transactions to be mined. Internal transactions are different than transactions that have been sent by the users, and the user who calls a function that triggers transactions has to pay gas fees for triggered transactions as well. We have run into this problem while calling the verify function in the sales workflow, and while calling the claim function in donations workflow. This was due to the fact that these functions triggered internal transactions. This problem was not only hard to manage, but also hard to detect as the error messages received from the Ethereum network were not explanatory enough.

6 Conclusions

Example workflows demonstrated in this paper show how smart contracts can add value to business processes. They can be used to remove third parties from transactions, registering results, and minting NFTs automatically. Ethereum smart contracts can be integrated into workflows via Zeebe Clients that interact with an Ethereum node through web3j methods. In order to use the solution proposed in this paper to deploy contracts, a Java wrapper class must be generated from solidity source code first. Also, in order to interact with a deployed smart contract, its address, and function descriptions must be known. Overall, using smart contracts within workflows is not a straightforward task and requires extensive knowledge of blockchain platforms and Web3 development. This situation may be setting an entry barrier for organizations that is preventing them from adopting new business processes that are including smart contracts. Model driven engineering tools have been created and explored for development of blockchain applications in business process management. However, existing tools either only generated smart contract code and didn't provide any means of deployment and execution or were developed as blockchain-based execution engines on their own. Therefore, it is still a hard task for organizations to integrate smart contract-based tasks into their existing workflows.

7 Future Work

We believe there is potential for developing middleware between process engines and blockchain platforms to simplify the usage of smart contracts. The proposed middleware may act as a registry containing information about smart contracts such as their addresses, descriptions, and functions they include. With such middleware, users would not only keep track of the smart contracts they use, but also would easily update their smart contracts without having to worry about updating their processes, since their processes will be using the registry for retrieving contract addresses. Another improvement may be a no-code or low-code solution that allows users to create, deploy and interact with smart contracts through their process modeler. Another issue that can be addressed by middleware may be the adjustment of gas fees. Utilizing the power of the process engine, a service task may try again with an increased gas limit if the sent transaction is reverted by Ethereum. Same applies to wait times between events, they can be organized by the middleware. Also, middleware would hide communication details from process developers. This would make using smart contracts in business processes an easier task.

References

[1] AgilePointBPMS. *What Is a Process Engine?* Accessed March 29, 2022. URL: https://documentation.agilepoint.com/supportportal/docs/productdocumentation/05.00.0200/documentationlibrary/maps/overviewWorkflowEngine.html.

[2] Massimo Bartoletti and Livio Pompianu. "An Empirical Analysis of Smart Contracts: Platforms, Applications, and Design Patterns". In: *Financial Cryptography and Data Security*. Ed. by Michael Brenner et al. Cham: Springer International Publishing, 2017, pp. 494–509. ISBN: 978-3-319-70278-0.

[3] Vitalik Buterin. *Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform*. Accessed July 11, 2022. 2014. URL: https://ethereum.org/669c9e2e2027310b6b3cdce6e1c52962/Ethereum_Whitepaper_-_Buterin_2014.pdf.

[4] Camunda. *Camunda Website*. Accessed July 11, 2022. URL: https://camunda.com/.

[5] Camunda. *Zeebe's Architecture*. Accessed July 11, 2022. URL: https://docs.camunda.io/docs/components/zeebe/technical-concepts/architecture/.

[6] Michele Chinosi and Alberto Trombetta. "BPMN: An introduction to the standard". In: *Computer Standards Interfaces* 34.1 (2012), pp. 124–134. ISSN: 0920-5489. DOI: https://doi.org/10.1016/j.csi.2011.06.002. URL: https://www.sciencedirect.com/science/article/pii/S0920548911000766.

[7] ConsenSys. *Over 350K Web3 Developers Now Use Blockchain Development Platform Infura*. Accessed July 11, 2022. URL: https://consensys.net/blog/press-release/over-350k-web3-developers-now-use-blockchain-development-platform-infura/.

[8] Pedro Costa. *Implementing a blockchain oracle on Ethereum.* Accessed July 11, 2022. 2019. URL: https://medium.com/@pedrodc/implementing-a-blockchain-oracle-on-ethereum-cedc7e26b49e.

[9] IBM Cloud Education. *What Is Workflow Automation?* Accessed March 29, 2022. URL: https://www.ibm.com/cloud/blog/workflow-automation.

[10] Ethereum. *Compiling Smart Contracts.* Accessed July 11, 2022. URL: https://ethereum.org/en/developers/docs/smart-contracts/compiling/.

[11] Ethereum. *Deploying Smart Contracts.* Accessed July 11, 2022. URL: https://ethereum.org/en/developers/docs/smart-contracts/deploying/.

[12] Ethereum. *What is Web3?* Accessed July 11, 2022. 2022. URL: https://ethereum.org/en/web3/.

[13] Investopedia. *What is a Blockchain?* Accessed March 29, 2022. URL: https://www.investopedia.com/terms/b/blockchain.asp#toc-what-is-a-blockchain.

[14] Preethi Kasireddy. *How does Ethereum work, anyway?* Accessed July 11, 2022. 2017. URL: https://www.preethikasireddy.com/post/how-does-ethereum-work-anyway.

[15] Victor Youdom Kemmoe et al. "Recent Advances in Smart Contracts: A Technical Overview and State of the Art". In: *IEEE Access* 8 (2020), pp. 117782–117801. DOI: 10.1109/ACCESS.2020.3005020.

[16] Satoshi Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System.* Accessed May 27, 2022. 2009. URL: https://bitcoin.org/bitcoin.pdf.

[17] Orlenys Pintado et al. "CATERPILLAR: A Business Process Execution Engine on the Ethereum Blockchain". In: (July 2018).

[18] Christoph Prybila et al. "Runtime verification for business processes utilizing the Bitcoin blockchain". In: *Future Generation Computer Systems* 107 (2020), pp. 816–831. ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2017.08.024. URL: https://www.sciencedirect.com/science/article/pii/S0167739X1731837X.

[19] Nick Szabo. *Smart Contracts.* Accessed July 11, 2022. 1994. URL: https://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart.contracts.html.

[20] Nick Szabo. *Smart Contracts: Building Blocks for Digital Markets.* Accessed July 11, 2022. 1996. URL: https://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart_contracts_2.html.

[21] An Binh Tran, Qinghua Lu, and Ingo Weber. "Lorikeet: A Model-Driven Engineering Tool for Blockchain-Based Business Process Execution and Asset Management". In: *BPM*. 2018.

[22] Web3j. *Deploy and Interact with Smart Contracts.* Accessed July 11, 2022. URL: https://docs.web3j.io/4.8.7/getting_started/deploy_interact_smart_contracts/.

[23] Web3j. *Web3j Documentation*. Accessed July 11, 2022. URL: https://docs.web3j.io/4.8.7/.

[24] Ingo Weber et al. "Untrusted Business Process Monitoring and Execution Using Blockchain". In: *Business Process Management*. Ed. by Marcello La Rosa, Peter Loos, and Oscar Pastor. Cham: Springer International Publishing, 2016, pp. 329–347. ISBN: 978-3-319-45348-4.