Blockchain-based Cross-Organizational Execution Framework for Dynamic Integration of Process Collaborations

Philipp Klinger[1], Freimut Bodendorf[1]

[1] Friedrich-Alexander-Universität Erlangen-Nürnberg, Lehrstuhl für Wirtschaftsinformatik, insb. im Dienstleistungsbereich, Nürnberg, Germany
{philipp.klinger,freimut.bodendorf}@fau.de

Abstract. Cross-organizational business processes involving multiple participants are choreographed, thus rely on mutual trust of collaborators or need to be coordinated by a central instance. Using Smart Contracts, business processes can be executed without a mutually trusted and centralized orchestrating authority. Former Blockchain-based execution framework proposals focus on orchestration diagrams as a basis for execution. Contrary, this work focuses on BPMN process collaboration diagrams as implementation basis and makes additional transformation steps obsolete. With the herein proposed framework for execution of cross-organizational process collaborations, another approach for the implementation and execution of inter-organizational processes on a Blockchain is presented, including a voting mechanism for process deployment as well as a subscription service to facilitate process handovers between participants more efficiently. The framework is exemplified and evaluated with a use case from a large German industrial manufacturing company.

Keywords: Business Process Management, Blockchain, Process Collaboration, Process Execution Engine, Ethereum

1 Motivation and Problem

Blockchain technology may serve for further use cases than just enabling the secure exchange of digital assets in a distributed peer to peer network. Some of these cross-organizational use cases can, for example, be seen in the production of and provenance of goods [1], logistics and supply chain [2], the energy markets [3], the public sector [4] or other highly regulated sectors [5].

In these inter-organizational processes, involving multiple process participants, the execution of business processes is often cumbersome as many different, to a large part incompatible frameworks, data-exchange protocols or computing systems are in use [2]. Traditional Business Process Engines cover the intra-organizational handling respectively orchestration of business processes well but offer limited support in handling mutual exchange of process data in multi-company settings [6]. With a

15th International Conference on Wirtschaftsinformatik,
March 08-11, 2020, Potsdam, Germany

https://doi.org/10.30844/wi_2020_i2-klinger

growing number of legally distinct organizations participating in a process, the interconnections between current Business Process Engines results in a combinatorial explosion in relation to the number of participants. Keeping all participants informed about the overall process state is a complex integration problem [7]. This is in practice solved following well-known B2B integration patterns, e.g., a Hub-and-Spoke architecture, Middleware- or Enterprise Service Bus solutions [2], [8].

Another problem of collaborative processes are adversarial settings in which participants do not trust each other to follow previously defined process steps [9, 10]. Therefore, trusted third parties (TTP) mediate non-trusting participants and verify actions or enforce the correct execution of a previously defined workflow. Downsides of a TTP in place are its compensation resulting in higher run-costs and additional operational delays or increased lay times during process execution.

Our contribution is an Ethereum-based process execution framework for cross-organizational process collaborations. The proposed solution solves the previously stated problems of integrating multiple parties in adversarial process settings. Smart Contracts are used as a mechanism to enforce a trusted and immutable process flow making the need for a TTP obsolete. Existing solutions focus on holistic on-chain execution of a multi-party process based on BPMN process orchestration diagrams [11–14]. Contrary, our proposed solution relies on BPMN collaboration diagrams as a starting point, without the need for additional transformations prior deployment. We also introduce an option to treat certain process participants or participating information systems (IS) like black boxes introducing a *Subscription Service*. This option is in line with our belief, that eventually execution of only adversarial process segments is needed on-chain, while other process collaborators, e.g. another IS or legacy process engine, in a collaboration model may be treated as private and only should react on defined inputs and communicate processed outcomes. The *Subscription Service*, as well as a multi-signature voting contract, are presented as a means to distribute run- and deployment costs fairly amongst participants.

Core concepts are introduced in section 2. Artifact design and implementation details will be given in sections 3 and 4. After a quantitative evaluation and comparison to related work, we conclude our findings and present future research opportunities.

2 Background

The Blockchain as a concept was first introduced by the cryptocurrency Bitcoin [15]. A Blockchain is a software system that acts as a distributed ledger in a peer to peer network of nodes that run a client software defining the Blockchain protocol. Through a distributed consensus mechanism that is part of the protocol specification, the peers of a network can synchronize with each other and create a universal truth in the form of a shared, transparent and consistent history of transactions. To store and communicate the transactions of the network efficiently, blocks are used as a data structure carrying the transactions. As the name Blockchain suggests, each newly added block is linked and also secured by making use of a cryptographic hash

https://doi.org/10.30844/wi_2020_i2-klinger

function on the contents of the block. Changes in the history of transactions become detectable and the system is, therefore, tamper-proof [16].
A Smart Contract is a program that resides as bytecode on the Ethereum Blockchain. These contracts are usually written in Solidity or other higher-order programming languages that may be compiled to Ethereum Virtual Machine (EVM) compatible instructions. Next to transferring value, an Ethereum transaction may additionally carry data payload to deploy a Smart Contract or invoke a state-change in the EVM by executing functions. Transactions are the only way to perform state changes in Smart Contracts. These state changes may be triggered by EOAs (Externally Owned Account) or other contracts, whereas all contract-initiated actions can always be traced back to an originating transaction issued by an EOA, thus a person or a machine owning a private key used to sign the initiating transaction. Once a transaction is mined into a block, state change computations will be performed by the EVM and the resulting transaction receipt is recorded on-chain. Also, corresponding event data is stored in an Ethereum Block, if instructed by a contract. Events can be used to log arbitrary data on the Ethereum Blockchain. Every EVM execution operation, e.g., performing computations, storing values or emitting events, is priced with gas, an Ethereum internal calculation unit. The total sum of gas for all operations executed multiplied by the current gas price of the Blockchain network results in the total price in Ethers that needs to be paid to execute those operations respectively transactions triggering the execution. In consequence, excessively expensive operations like the creation of new contracts, the use of storage and therefore gas-intensive writing operations should be used sparsely to keep execution as well as deployment costs on a minimum. [17]

3 Conceptual Design

The presented work on the following execution framework is part of a larger effort to implement a decentralized Business Process Management System based on Ethereum. Other components of the umbrella framework, like the graphical user interface for convenient process execution and monitoring, will not be covered in this paper. Our framework proposes a solution to the challenges of cross-organizational collaboration in adversarial settings to collaboratively execute *all* or *only parts of* a highly structured process using Smart Contracts without the need of a TTP.

3.1 Design Decisions

Following design choices characterize our approach (cf. section 5.3 for related work):

Simplicity of process to contract conversion: Contrary to [12, 13] we create contracts from a BPMN collaboration diagram without the need for a previous conversion from a collaboration to a single-pool (process orchestration) representation.

The collaboration diagram-based approach is straight-forward and does not yield the possibility to confuse process stakeholders with different process visualizations to

https://doi.org/10.30844/wi_2020_i2-klinger

the ones used for implementation. In addition, the single-pool approach is ambiguous in expressing responsibilities and handoffs between collaborators in big process models and bears the risk of process inflexibility due to non-upgradeable contracts implementing the process [18]. Our collaboration-based approach enables flexible integration of process segments that are handled on-chain without the need of a TTP as well as off-chain process segments, that may be implemented by traditional enterprise systems to interact with deployed process contracts on the Blockchain. On the other hand, in the single-pool approach, all participants must interact on-chain without the possibility of any process changes after deployment.

No single authority: Whereas [12–14] rely on an administrative entity to deploy process contract(s) during build-time, our proposed solution builds upon a non-authoritative voting-based system. It is used to dynamically deploy and subsequently administer roles and access rights during run-time. Changes to initial process settings require agreement by all entitled participants of the process in the form of proposals that must be signed by all participants or a previously defined threshold of process participants. The voting contract, once set up, is used to pay expenses as an EOA for subsequent contract proposals respectively proposal executions.

Fair cost distribution: As there is no single authority deploying the process contract(s) as in [10–13], costs must be distributed fairly amongst collaborators for the deployment through the submission of a contract creation proposal in the *Voting Contract* which must be signed by the collaborators before taking effect and execution.

Generally, costs split up in deployment costs for the shared, static part of the execution framework, and deployment costs for the dynamic contract components that build up the process definition. In an optimal setting, runtime costs for the execution of process instances are paid for by every participant in relation to his executed activities.

Optional subscription service: To integrate off-chain enterprise software such as legacy workflow systems or BPMS, a *Subscription Service* similar to the "active mediator" [13] can optionally be installed by a process participant to manage automatic workflow handovers. While frameworks rely on a centrally installed component [12, 13, 19]. In our framework, no central component is needed, e.g. message-receiving participants may either automatically react to the message-handoff by using the optional *Subscription Service* watching respective message events or react manually, without such a service running, to the handoff. Each participant may optionally install the NodeJS-based *Subscription Service* (locally) to react upon events emitted by direct communication partners. This helps to reduce lay times and attributes to fair cost distribution amongst participants during the execution of contracts.

Process logic as an auditable state machine: For every participant respectively pool in a collaboration diagram one process contract will be instantiated. In order to transform process flow to Solidity, states of the workflow must be known (c.f. section 4.2) to create a finite state machine representation of the workflow [20].

Process auditability: During process execution, we write actions of noticeable events to the Ethereum event log, e.g. when an activity was successfully performed or

https://doi.org/10.30844/wi_2020_i2-klinger

to indicate handoffs to other participants. This is similarly done by other frameworks like [12–14]. The logs form an auditable execution trail that may be analyzed using process mining [21, 22]. Other frameworks, like [12, 19], rely on centrally installed components while our framework does not rely on a central component. A message-receiving participant may either automatically react to the message-handoff by using the optional *Subscription Service* watching respective message events or react manually, without such a service running, to the handoff.

Network agnostic framework: Our framework is designed to run in public as well as private Ethereum Blockchain network environments. Depending on the use case, one may choose a private network over a public network to avoid data-privacy issues in the first place when transacting with sensitive personal or process data.

3.2 Execution Framework Contract Design

Components of our execution framework may be logically separated into static and dynamic components as shown in Fig 1.

Once collaborators decide to transact in collaborative processes as a consortium on the Blockchain, **static components (I)** serve as a base framework layer. These contracts are deployed just once per execution framework instance and mainly offer management functionality, e.g., for keeping track of consecutively deployed process definitions, managing participants' access and execution rights or voting for new process deployment proposals through a multi-signature voting contract.

The **dynamic components (II)** need to be executed and deployed per collaborative process definition (collaboration diagram) by the consortium. Static components of the framework give instructions on how dynamic components must be structured and therefore guarantee interoperability with the execution framework.

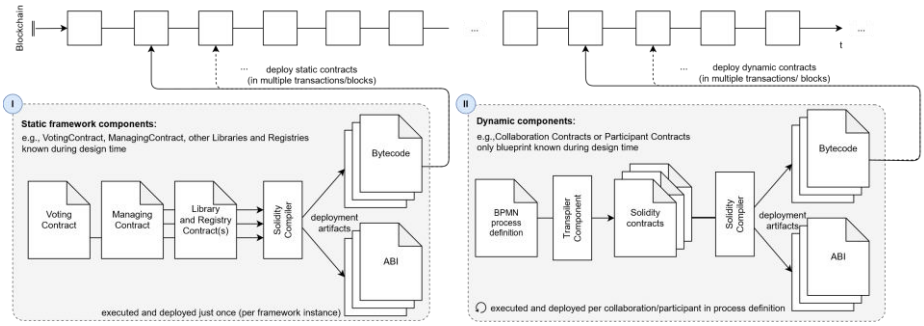

Fig 1. Architecture of the execution framework with static (I) and dynamic (II) components.

Based on previously stated design decisions (cf. section 3.1) and distinction in static or dynamic components, we propose the following contract design as shown in Fig 2.

Static components (I): The static components are just deployed once to the Blockchain per collaboration consortium and therefore behave like singletons.

https://doi.org/10.30844/wi_2020_i2-klinger

Contract libraries include common functionality, that is reused amongst more than one contract and help to reduce deployment costs. A segregated data storage contract is used to decouple contract logic from instance data payload [20].

All contracts inheriting from an *Abstract* contract copy all implemented functionality defined in the abstract contract and may extend this functionality fulfilling given method definitions (cf. Fig 2). Inheritance is used to standardize framework interfaces.

The *VotingContract* allows the creation of, voting on and execution of proposals. A proposal is an encoded transaction, that needs the agreement of several consortium members of the instantiated execution framework to be executed. Such a transaction may implicate the deployment of a new contract (contract creation transaction) or the execution of other contract functions (through a contract transaction) altering contract state. A contract creation transaction we make use of is the instantiation of the *ManagingContract*. Whereas a contract transaction would be an update of the *owner* address variable in the *ManagingContract.* This would be needed if a *VotingContract* reference inside the *ManagingContract* must be swapped out in favor of another owner address, i.e. another consortium, to accept orders from.

The main functionality of the *ManagingContract* is to store the contract address of the eligible *VotingContract* that may perform administrative actions and managing contract address references of *CollaborationContract*s.

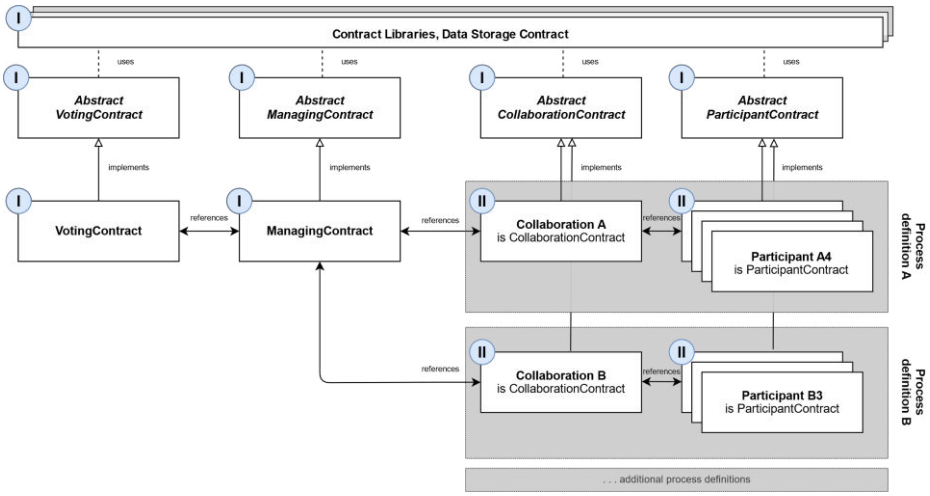

Fig 2. Contract design of the execution framework split into static (I) and dynamic (II) components (cf. Fig 1) with two transformed collaboration definitions (A and B)

Dynamic components (II): Each process definition leads to a separate deployment, that may be dynamically added to the framework during runtime. The *Transpiler* component (cf. Fig 1) parses the given BPMN for the collaboration identifier string, and names the resulting *CollaborationContract* (cf. Fig 3) accordingly. All functionality of the created Solidity contract e.g. functions, type definitions or events are inherited from an *AbstractCollaborationContract* template,

https://doi.org/10.30844/wi_2020_i2-klinger

thus guaranteeing interoperability with the other components of our execution framework. Each participant in the BPMN collaboration diagram will be transpiled into a *ParticipantContract* implementing state definitions (also see section 4.2) and genuine state management functionality as defined by the *AbstractParticipantContract* base template for each *ParticipantContract.* Contract address references of deployed *ParticpantContract* are managed in the corresponding *CollaborationContract*.

4 Prototype

4.1 Use Case

To showcase the feasibility of the hereby presented approach, a simplified BPMN collaboration is presented which expresses the handling of a multi-stage distribution channel process for spare parts handling of a large German electronics manufacturing company with worldwide distributed embassies in different countries (cf. Fig 3).

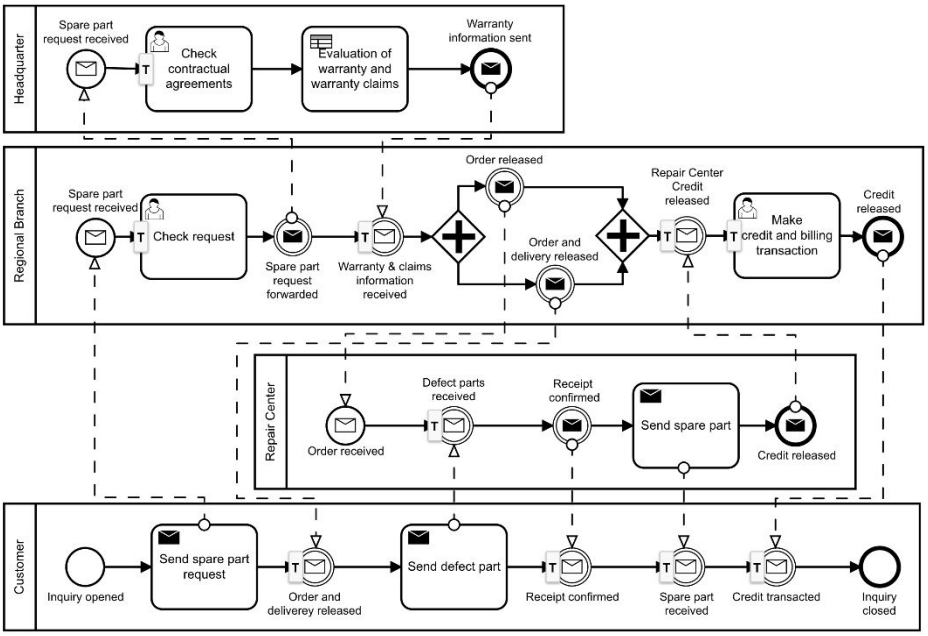

Fig 3. Simplified BPMN collaboration diagram of a spare parts process, stemming from a large German electronics manufacturing company, including transaction boundaries (T)

In the implemented legacy process at the electronics manufacturing company, standardization of communication interfaces and managing the interconnections of the actors involved in the process on a worldwide level are a constant problem. Since it is a customer-facing process, it must be optimized for process quality as well as fast

https://doi.org/10.30844/wi_2020_i2-klinger

throughput. This is currently a challenge due to inconsistencies in process enforcement across different regional branches as well as lacking transparency of the current process state, which in return results in high lay times.

4.2 Contract Interaction and State Management

Regarding the ***VotingContract***, we currently make use of a slightly adapted version of the Ethereum DAO with arbitrary bytecode execution functionality [23]. Using this *VotingContract*, a consortium of participants may, once assembled, vote on new proposals to execute (via its *executeProposal* method [24]) or create new process definitions as Smart Contract deployments. Other frameworks [11–14] rely on a central entity to administer and deploy new process models. Since all proposals need to be passed in as bytecode in our approach and need to be voted and accepted prior to execution, it naturally comes at greater costs than a supervised deployment. After the *VotingContract* is set up by an EOA in the first place, static and dynamic contracts can be deployed. Although static contracts can be set up as well through the *VotingContract*, its greatest perceived benefit lies in a cost-fair instantiation of new *Participant*- and *CollaborationContracts*. In the case example, four entities take part in voting.

The ***ManagingContract*** acts as a simple contract register [20, 24] to administer different process definitions. It saves *CollaborationContract* address references and provides lifecycle methods for the management of collaborations. A *ManagingContract* that administers *CollaborationContracts* is always owned by a *VotingContract* reference and may only be altered by the *VotingContract* address.

To keep track of all generated *ParticipantContract*s, the associated ***CollaborationContract*** also acts as a simple register enabling lookup and interaction between the participants. Also, meta-information supplementing the process definition is stored in the *CollaborationContract*, e.g., the current process version to the respective contract deployment addresses and a hash value linking the corresponding BPMN file used to generate the contracts as a reference [24]. The *CollaborationContract* implementation also keeps track of *process instances*, their creation, and its logging.

From the given process model in Fig 3, using the *Transpiler,* we derive one *CollaborationContract* and one *ParticipantContract* per given pool in the diagram: *Customer*, *RepairCenter*, *RegionalBranch,* and *Headquarter*.

Every concrete ***ParticipantContract*** keeps track of its own states as an array of bytes32-values (cf. 5.1). In order to derive the states, our approach is to align process states with BPMN transaction boundaries. The result is an array of *wait states* expressing places where process instance tokens can reside. In our adopted definition, *wait states* are Receive- or User tasks, Message-, Timer- or Signal-events and the Event-based Gateway in BPMN [25] and are separated by transaction boundaries (T). The number of wait states in a pool is thus: the number of derived transaction boundaries T per pool plus one. Our NodeJS-based *Transpiler* module (cf. Fig 1 and Fig 4) parses the states from a given BPMN file according to rules defined by [25] and builds the state transitioning functions in every *ParticipantContract* for the

https://doi.org/10.30844/wi_2020_i2-klinger

respective *wait states* derived. For the *RegionalBranchContract*, five states are derived, which are stored in a bytes32 array. The values of this array are extracted by the Transpiler, in our case the element's id attribute prior to a subsequently identified wait state. Deriving unique identifiers that correlate with the BPMN elements is crucial for highlighting process flow in a frontend and to generate meaningful log statements.

Choosing the execution steps according to the wait-state approach is beneficial, since Ethereum transactions may fail due to various reasons. This may happen, e.g., if requirements during contract execution are not fulfilled and the contract initiates a "throw" to end execution or if other internal failures like an out of gas error occur when a maximum execution depth is reached. In our execution framework, the wait state transitions performed by Ethereum state transitions through transactions directly mimic ACID properties of transactions of RDBMS used for process state transitioning of common process engines. If a process engine would fail in case of failure during a state transition, i.e. a service is unreachable or a message cannot be sent, the process state rolls back to the initial state and a retry will be scheduled. Similarly, on processing failures in Ethereum, the rollback is a default behavior handled by the EVM.

As the states of the process model are identifiable with the method described above, a state machine representation of the workflow can be created. For every wait state derived from the process model, one state transition function must be implemented in the respective *ParticipantContract*. State transition functions are named as encoded in the states array and check the entrance eligibility by comparing current state with the state required for execution using Solidity function modifiers. Additionally, execution capability is checked rudimentarily using another allowance modifier. If any checks fail, the EVM reverts and the state does not progress. If all checks resolve, the wait state payload is executed, e.g., performing calculations, making calls to other contracts. Afterward, the current state is logged emitting a Solidity event for traceability. Lastly, another modifier will alter the state to the next eligible state and the previously described procedure recommences with the next process (wait) state.

4.3 Event Logging and Event Subscription Service

If an activity or task is performed by a participant, the state of the process is logged emitting an event. This concept resembles the event log of traditional process engines, whereas the event logs are stored in the Ethereum Blockchain block-structure. The log structure used by our implementation comprises the event emitter (participant address), the collaboration contract address, a process instance ID, the activity performed, as well as the user identifying wallet address that triggered the activity. Additionally, the block number and thus the (rough) execution time may be recovered from the block header.

All emitted and persisted events of a process combined form an audit trail on the Blockchain that is retrieved using a watcher script running on a local Ethereum client. Watchers can be installed listening to certain emitting addresses or topics filtering for

https://doi.org/10.30844/wi_2020_i2-klinger

the retrieval of events in a certain range of blocks, all blocks, or only newly created blocks. This approach resembles a Publish/Subscribe messaging pattern [8] often used to decouple systems. In a similar fashion, this pattern decouples process participants in our framework. Our *Subscription Service*, which is an optional software component offered by our execution framework, lets a participant react to emitted Ethereum events and trigger own actions that resemble the receiving of message events. Instead of having to invocate transactions manually to progress state, one is now able to check if a wait state condition has resolved to continue process flow. This concept also enables the usage of black-box participants, which only react to incoming and trigger outgoing messages in the form of Ethereum events, that other process participants can subscribe and react to. Waiting times for (human) process progression can be minimized and transaction costs (for EVM operations performed such as state update) are paid only by the executing participant for the state changes performed.

To enable this concept, the triggers (subscription topics) must be known by the *Subscription Service*. We extract relevant message flows (event topics) directly from the given BPMN collaboration diagram using a parser script and store the results in an artifact repository of the executing participant. The *Subscription Service* takes this parsed input as a NodeJS-based command line script. In order to extract events, the subscription service must have access to an Ethereum full node storing the events. The *Subscription Service* is an optional component. Each instance (per participant) needs to hold a private key to sign transactions automatically for interactions with other process participants.

To sum up, all previously described components, as well as their interactions with other components following sections 3 and 4, are represented in the general architectural overview (cf. Fig 4).

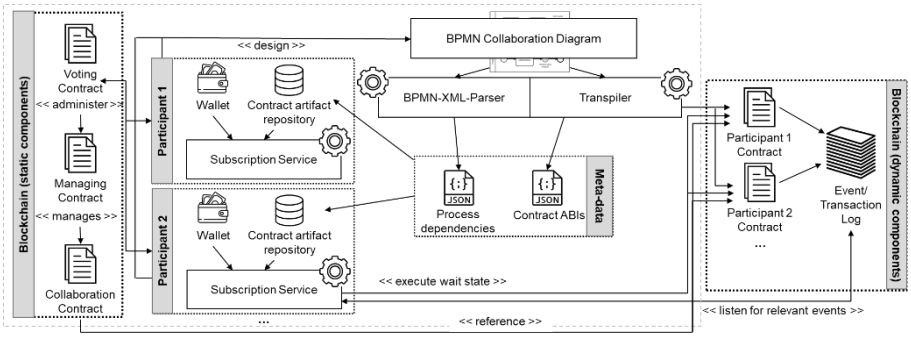

Fig 4. General architectural overview

https://doi.org/10.30844/wi_2020_i2-klinger

5 Evaluation

5.1 Cost Analysis

Currently, our main goal is to develop a working prototype to validate the design and test the practicality of the execution framework implementation in a private network. Accordingly, the prototype is not cost-optimized, i.e., bytes32 identifiers are used to log performed activities that could be improved to save on incurred storage costs. If cost-optimization is a major concern, all (wait) states can be encoded in integer variables [26] at the expense of convenience and need for an additional mapping construct to convert encodings when dealing with process event logs.

Cost calculations were performed in a local test network. Gas price is constant at 20 Gwei. To give a rough price estimation for execution in a public network, the price of 1 ETH at the time of writing is assumed to be USD $230.

In the cost breakdown provided (cf. Table 1) contract deployment and run costs are presented, including costs for one-time setups, e.g., to initialize allowance contract addresses. Dynamic components except the *CollaborationContract* do not need any extra setup after contract deployment. In those cases, only run costs are considered.

The costs of the adopted *VotingContract* is highest amongst all contracts. As the *VotingContract* costs are incurred only once and since it can be used managing many process collaborations, the costs are arguable. Nevertheless, the *VotingContract* implementation needs to be optimized.

One can see that running the process collaboration *without* the *Subscription Service*, costs are irregularly distributed among participants since triggering participants would pay transaction costs for process handovers. Consequently, this leads to unfair cost distribution in public networks but could be tolerated in a private network. *With* the *Subscription Service* running, costs are fairly distributed amongst participants in relation to the actions performed in their respective pools while overall cost stays constant. Therefore, running the process in a public network would be favorable with the *Subscription Service* regarding execution costs.

Table 1. Deployment, setup and run costs with or without Subscription Service running

Contract	***Deployment Cost***		***Additional Setup Cost***		***Run Cost per process instance with(out) Subscription Service***			
Static (I)	**ETH**	**USD**	**ETH**	**USD**	**ETH**	**USD**	**ETH**	**USD**
VotingContract	0,06108	14,05	0,00243	0,56	-	-	-	-
ManagingContract	0,01984	4,56	0,00127	0,29	-	-	-	-
Dynamic (II)					**without**		**with**	
SPP_Collab.Contract	0,01528	3,51	0,00215	0,49	-	-	-	-
SPP_Customer	0,02304	5,30	-	-	0,00206	0,47	0,00454	1,04
SPP_Headquarter	0,01424	3,28	-	-	0,00633	1,46	0,00182	0,42
SPP_RegionalBranch	0,01915	4,40	-	-	0,00373	0,86	0,00395	0,91
SPP_RepairCenter	0,01606	3,69	-	-	0,00000	0,00	0,00182	0,42
Sum	0,16870	38,80	0,00586	1,35	0,01213	2,79	0,01213	2,79

https://doi.org/10.30844/wi_2020_i2-klinger

5.2 Limitations and Future Improvement Potential

Currently, our *Transpiler* component produces an executable state-machine Smart Contract from the given BPMN collaboration. Yet, for some modeling constructs, such as events, complex or event-based gateways, logic needs to be added manually to the respective *ParticipantContracts* prior deployment. Other comparable execution engines on the Ethereum Blockchain already cover a large part of the BPMN palette [12]. This effort will be a follow-up step to enhance our proposed execution framework and to be able to trigger fully automated contract deployments with only a collaboration diagram given in the first place.

Currently, our framework is not focusing on data privacy aspects since we are operating in a private network. Once a concept to deal with (personal and processual) data in a public network is implemented, the framework can be ported to a public environment. The implementation is portable to EVM-compatible Blockchain systems like Hyperledger Burrow, Quorum or Parity with privacy features enabled.

In a public network, scalability issues may arise, e.g., due to high gas costs or as of network capacity. Additional performance optimizations to reduce gas usage in the logic contracts will improve the implementation in the next iteration.

Another future improvement to our approach is better support for upgradeability, e.g., versioning and execution of versioned processes. This includes the introduction of a proxy contract holding references to the current implementation contract.

5.3 Comparison to Related Work

Different proposals towards process execution or verification using a Blockchain system exist. Authors of [9] use Bitcoin's mechanisms to enforce and verify process flow, removing the need for mutual trust in contractual relationships. Since Bitcoin's technical possibilities are limited [9], we consider Ethereum as an implementation platform mainly due to its versatile Smart Contracting possibilities, development tool availability and widespread adoption.

Other approaches propose domain-specific language constructs like DCR-graphs [10] or Business Artifacts [27] as a starting point for Blockchain-based process execution frameworks. Authors of [14] propose a lightweight execution framework for cross-organizational process execution on Ethereum entirely on-chain. Characteristic of the latter approach is the absence of a process model during contract deployment, thus the process model will be defined after deployment during runtime [14]. This is beneficiary for efficient scaling, yet in our opinion unsuitable for changing processes creating the need for frequent, cost-intensive, contract deployments.

Early efforts comparable to our framework already show that process descriptions in the form of BPMN 2.0 files can be translated into a codified contractual enforcement of the process flow in a Solidity Smart Contract which is able to manage process state and orchestrate process execution on Ethereum while simultaneously documenting performed process steps on-chain [11],[13]. Follow-up work focuses on contract performance optimizations using Petri Nets during BPMN transpilation and

https://doi.org/10.30844/wi_2020_i2-klinger

other Solidity related performance measures like using bit vectors to reduce execution- and storage costs [26]. As stated in section 5.2, our presented framework is currently unoptimized and relies on Petri net verification enabled by external tools prior to state machine generation and relies on the presented wait-state approach per participant respectively pool.

Another model-driven engineering tool [19] and a food traceability system [28] make use of the process execution approach originally proposed in [13] to manage and track process state. A business process management system, called Caterpillar, is initially proposed by [11]. According to Caterpillar's first design principle, the collaborative process in form of a BPMN collaboration diagram must be transformed into an orchestration diagram where each participant is modeled as a swim lane and handoffs between (external) participants are handled with a sequence flow passing lanes like in an internal orchestration diagram rather than using messages as an exchange mechanism [12]. This need for an extra conversion step from a cross-organizational process diagram in the form of a collaboration to an intra-organizational diagram in the form of a single-pool orchestration may confuse stakeholders dealing with the process during execution. The referred design principle stated by [12] contrasts our proposed execution framework, as the basis for our proposed process engine is a collaboration diagram, that does not need to be transformed to an orchestration diagram prior to compilation. We rely on existing approaches by [20, 29, 30] to transform a process model to a state machine representation.

Most recent work is also using Ethereum for execution of collaborative workflows based on an extended BPMN standard for choreographies [31]. While their solution is comparable to ours to the extent of process execution and verification, their implementation lacks "provisions to evenly distribute costs between participants" [31]. Contrasting, in our implementation, we may clearly separate cost at the pool boundaries with the help of the *Subscription Service*. Also, our solution sticks to established modeling standards to make use of the existing BPMN toolchain.

None of the related works is making use of a multi-signature voting mechanism during deployment phase to distribute costs, but rather follow approaches, where one central entity is responsible for process deployment.

7 Conclusion

Our proposal for a cross-organizational Blockchain-based process execution framework solves integration and trust problems that commonly arise with many participants that need to collaborate on a commonly defined workflow. The framework provides process transparency to all process participants thus having the potential to reduce costly lay times and quality improvements of strictly enforced workflows. Also, the need for a trusted third party overlooking a process is eliminated, through contract-based process execution. Contrasting other execution frameworks, we take a BPMN collaboration diagram as basis for process implementation. Static and dynamic framework components enable flexible integration of contract-based collaborations during runtime.

https://doi.org/10.30844/wi_2020_i2-klinger

A transpiler is used to automatically generate Smart Contracts resembling the process flow based on a BPMN collaboration diagram. A voting system ensures fair distribution of deployment costs amongst participants. An optional *Subscription Service* is presented, that may be used to reduce waiting times when executing the process and fosters fair cost-attribution amongst the collaborators during runtime.

Another unique distinction of our framework is the support for collaborations in which only a subset of participants needs to be covered 'on-chain' as Smart Contracts, whilst others may remain 'off-chain' and communicate through events with the process segments implemented as Smart Contracts. Thus, offering the possibility to only treat certain participants or shared process segments on-chain and have other participants defined as 'black boxes'.

References

1. Hackius, N., Petersen, M.: Blockchain in logistics and supply chain. In: Proceedings of the Hamburg International Conference of Logistics (HICL), pp. 3–18. epubli GmbH, Berlin (2017)
2. Korpela, K., Hallikas, J., Dahlberg, T.: Digital Supply Chain Transformation toward Blockchain Integration (2017)
3. Albrecht, S., Reichert, S., Schmid, J., Strüker, J., Neumann, D., Fridgen, G.: Dynamics of Blockchain Implementation - A Case Study from the Energy Sector. In: Bui, T. (ed.) Proceedings of the 51st Hawaii International Conference on System Sciences. Hawaii International Conference on System Sciences (2018)
4. Ølnes, S., Jansen, A.: Blockchain technology as infrastructure in public sector. In: Janssen, M., Chun, S.A., Weerakkody, V. (eds.) Proceedings of the 19th Annual International Conference on Digital Government Research Governance in the Data Age - dgo '18, pp. 1–10. ACM Press, New York, New York, USA (2018)
5. Fridgen, G., Radszuwill, S., Urbach, N., Utz, L.: Cross-Organizational Workflow Management Using Blockchain Technology - Towards Applicability, Auditability, and Automation (2018)
6. Mendling, J., Weber, I., van der Aalst, W., vom Brocke, J., Cabanillas, C., Daniel, F., Debois, S., Di Cicco, C., Dumas, M., Dustdar, S., et al.: Blockchains for Business Process Management - Challenges and Opportunities. ACM Trans. Manage. Inf. Syst. 9, 1–16 (2018)
7. Xu, L., Liu, H., Wang, S., Wang, K.: Modelling and analysis techniques for cross-organizational workflow systems. Syst. Res. 26, 367–389 (2009)
8. Hohpe, G., Woolf, B., Brown, K.: Enterprise integration patterns. Designing, building, and deploying messaging solutions. Addison-Wesley, Boston, Mass. (2012)
9. Prybila, C., Schulte, S., Hochreiner, C., Weber, I.: Runtime verification for business processes utilizing the Bitcoin blockchain. Future Generation Computer Systems (2017)
10. Madsen, M.F., Mikkel Gaub, Tróndur Høgnason, Kirkbro, M.E., Tijs Slaats, Søren Debois (eds.): Collaboration among Adversaries: Distributed Workflow Execution on a Blockchain (2018)
11. López-Pintado, O., García-Bañuelos, L., Dumas, M., Weber, I.: Caterpillar: A blockchain-based business process management system. In: Robert Clarisó, Henrik Leopold, Jan Mendling, Wil M. P. van der Aalst, Akhil Kumar, Brian T. Pentland, and Mathias Weske

https://doi.org/10.30844/wi_2020_i2-klinger

(ed.) Proceedings of the BPM Demo Track (BPM'17), 1920 (2017)
12. López-Pintado, O., García-Bañuelos, L., Dumas, M., Weber, I., Ponomarev, A.: CATERPILLAR: A Business Process Execution Engine on the Ethereum Blockchain (2018)
13. Weber, I., Xu, X., Riveret, R., Governatori, G., Ponomarev, A., Mendling, J.: Untrusted Business Process Monitoring and Execution Using Blockchain. In: La Rosa, M., Loos, P., Pastor, O. (eds.) Business Process Management, 9850, pp. 329–347. Springer International Publishing, Cham (2016)
14. Sturm, C., Szalanczi, J., Schönig, S., Jablonski, S.: A Lean Architecture for Blockchain Based Decentralized Process Execution. In: Daniel, F., Sheng, Q.Z., Motahari, H. (eds.) Business Process Management Workshops, 342, pp. 361–373. Springer International Publishing, Cham (2019)
15. Nakamoto, S.: Bitcoin: A Peer-to-Peer Electronic Cash System, https://bitcoin.org/bitcoin.pdf
16. Narayanan, A., Bonneau, J., Felten, E., Miller, A., Goldfeder, S.: Bitcoin and cryptocurrency technologies. A comprehensive introduction. Princeton University Press, Princeton (2016)
17. Wood, G.: Ethereum: A secure decentralized generalised transaction ledger. Byzantinum Version, https://ethereum.github.io/yellowpaper/paper.pdf
18. Zhang, P., White, J., Schmidt, D.C., Lenz, G.: Design of blockchain-based apps using familiar software patterns with a healthcare focus. In: Proceedings of the 24th Conference on Pattern Languages of Programs, pp. 1–14. The Hillside Group, USA (2017)
19. Tran, B. an, Lu, Q., Weber, I.: Lorikeet: A Model-Driven Engineering Tool for Blockchain-Based Business Process Execution and Asset Management. In: Wil M. P. van der Aalst, Casati, F., Conforti, R., Leoni, M.d., Dumas, M., Kumar, A., Mendling, J., Nepal, S., Pentland, B.T., Weber, B. (eds.) Business Process Management 2018, pp. 56–60 (2018)
20. Wöhrer, M., Zdun, U.: Design Patterns for Smart Contracts in the Ethereum Ecosystem. In: 2018 IEEE International Conference on Blockchain (2018)
21. Klinkmüller, C., Ponomarev, A., Tran, A.B., Weber, I., van der Aalst, W.: Mining Blockchain Processes: Extracting Process Mining Data from Blockchain Applications. In: Di Ciccio, C., Gabryelczyk, R., García-Bañuelos, L., Hernaus, T., Hull, R., Indihar Štemberger, M., Kő, A., Staples, M. (eds.) Business Process Management 2019, 361, pp. 71–86. Springer International Publishing, Cham (2019)
22. Mühlberger, Roman, Stefan Bachhofner, Claudio Di Ciccio, Luciano García-Bañuelos, and Orlenys: Extracting Event Logs for Process Mining from Data Stored on the Blockchain. In: Second Workshop on Security and Privacy-enhanced Business Process Management (SPBP), BPM Workshops. Vienna, Austria (2019)
23. Decentralized Autonomous Organization. How to build a democracy on the blockchain, https://www.ethereum.org/dao
24. Xu, X., Pautasso, C., Zhu, L., Lu, Q., Weber, I.: A Pattern Collection for Blockchain-based Applications. In: ACM (ed.) Proceedings of the 23rd European Conference on Pattern Languages of Programs - EuroPLoP '18, pp. 1–20. ACM Press, New York, New York, USA (2018)
25. Camunda: Transactions in Processes. Wait States, https://docs.camunda.org/manual/7.5/user-guide/process-engine/transactions-in-processes/#wait-states
26. García-Bañuelos, L., Ponomarev, A., Dumas, M., Weber, I.: Optimized Execution of

https://doi.org/10.30844/wi_2020_i2-klinger

Business Processes on Blockchain. In: Carmona, J., Engels, G., Kumar, A. (eds.) Business Process Management, 10445, pp. 130–146. Springer International Publishing, Cham (2017)
27. Hull, R., Batra, V.S., Chen, Y.-M., Deutsch, A., Heath III, F.F.T., Vianu, V.: Towards a Shared Ledger Business Collaboration Language Based on Data-Aware Processes. In: Sheng, Q.Z., Stroulia, E., Tata, S., Bhiri, S. (eds.) Service-Oriented Computing, 9936, pp. 18–36. Springer International Publishing, Cham (2016)
28. Lu, Q., Xu, X.: Adaptable Blockchain-Based Systems: A Case Study for Product Traceability. IEEE Softw. 34, 21–27 (2017)
29. Mavridou, A., Laszka, A.: Designing Secure Ethereum Smart Contracts: A Finite State Machine Based Approach. FC 2018 10957, 523–540
30. Xu, X., Weber, I., Staples, M.: Architecture for Blockchain Applications. Springer International Publishing, Cham (2019)
31. Ladleif, J., Weske, M., Weber, I.: Modeling and Enforcing Blockchain-Based Choreographies. In: Hildebrandt, T., van Dongen, B.F., Röglinger, M., Mendling, J. (eds.) Business Process Management, 11675, pp. 69–85. Springer International Publishing, Cham (2019)

https://doi.org/10.30844/wi_2020_i2-klinger