Using Blockchain to Enable Untrusted Business Process Monitoring and Execution

Ingo Weber[1,2] Xiwei Xu[1,2] Régis Riveret[1]
Guido Governatori[1] Alexander Ponomarev[1] Jan Mendling[3]

[1] Data61, CSIRO, Australia
{firstname.lastname}@data61.csiro.au
[2] University of New South Wales, Sydney, Australia
[3] Wirtschaftsuniversität Wien, Vienna, Austria
jan.mendling@wu.ac.at

Technical Report
UNSW-CSE-TR-201609
June 2016

THE UNIVERSITY OF
NEW SOUTH WALES

School of Computer Science and Engineering
The University of New South Wales
Sydney 2052, Australia

Abstract

The integration of business processes across organizations is typically beneficial for all involved parties. However, the lack of trust is often a roadblock. *Blockchain* is an emerging technology for decentralized and transactional data sharing across a network of untrusted participants. It can be used to find agreement about the shared state of collaborating parties without trusting a central authority or any particular participant. Some blockchain networks also provide a computational infrastructure to run autonomous programs called *smart contracts*. In this paper, we address the fundamental problem of trust in collaborative process execution using blockchain. We develop a technique to integrate blockchain into the choreography of processes in such a way that no central authority is needed, but trust maintained. Our solution comprises the combination of an intricate set of components, which allow monitoring or coordination of business processes. We implemented our solution and demonstrate its feasibility by applying it to three use case processes. Our evaluation includes the creation of more than 500 smart contracts and the execution over 8,000 blockchain transactions.

1 Introduction

The integration of business processes, e.g., along the supply chain, has been found to contribute both to better operational and business performance [4, 10]. A lack of trust, however, may hamper the innovativeness of further developing the collaborative process and its performance altogether [13]. Once service-level agreements are in place, it becomes a highly delicate question which partner should serve as a hub for controlling the collaborative process of several parties, or where a mediator process is hosted. While control asymmetries can be avoided by a decentralized choreography instead of central orchestration, it does not solve the general problem of trust in controlling the collaborative business process.

The described lack-of-trust problem can be addressed with novel blockchain technology. Instead of agreeing on one trusted party, participants share transactional data across a large network of *untrusted* nodes (i.e., machines). This is achieved using a timestamped list of blocks which record, share, and aggregate data about transactions that have ever occurred within the blockchain network. Cryptographic proofs make this data storage immutable. As long as a majority share of the blockchain is not compromised, transactions can only be inserted; updating or deleting existing transactions is prohibitively expensive, making the blockchain tamper-proof. Blockchain also provides a global computational infrastructure, which can run programs: so-called *smart contracts* [12] execute across the blockchain network and automatically enforce the conditions defined in the transactions to enable, for example, conditional payment.

In this paper, we adopt blockchain technology to address the lack-of-trust problem in collaborative business processes. More specifically, we develop an approach to map a business process onto a peer-to-peer execution infrastructure that stores transactions in a blockchain, offering the following benefits. First, we provide a monitoring facility that integrates an automatic and immutable transaction history. Second, smart contracts can be used as a direct implementation of the mediator process control logic. Third, we obtain an audit trail for the complete collaborative business processes, for which payments, escrow, and conflict resolution can be enforced automatically. Our contribution is the first approach and implementation that leverages blockchain for collaborative process execution and monitoring. We evaluate our approach for feasibility by prototyping three use case processes on top of it. To this end, we ran of more than 500 process instances by creating as many smart contracts, and executed over 8,000 blockchain transactions that interact with the smart contracts.

The paper proceeds with a discussion of the research problem, related work, and blockchain technology in Section 2. Section 3 presents the details of our approach. Section 4 evaluates our approach using several real-world business scenarios, and Section 5 concludes. This technical report serves as long version of a conference paper [23]. Finally, a screencast video is available.[1]

2 Background

This section discusses the research problem we address, related work, and the background of blockchain technology as a solution.

[1] https://youtu.be/1SNn9c5HHQs

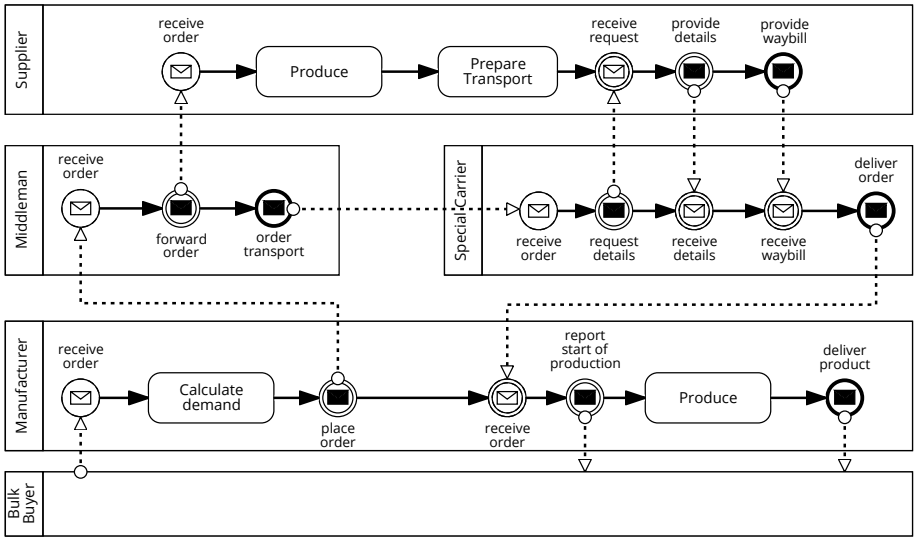

Figure 2.1: Supply Chain Scenario from [3] (simplified)

2.1 Challenges of Collaborative Business Process Execution

We illustrate challenges of executing collaborative business processes by the help of a supply chain scenario reported in [3] that we simplify in Fig. 2.1. The process starts with the Bulk Buyer placing an order with the Manufacturer. The latter calculates the demand and places an order for materials via a Middleman. This Middleman forwards the order to a Supplier and arranges transportation by a Special Carrier. Once the materials are produced, the Carrier picks them up at the Supplier site and delivers them to the Manufacturer. The Manufacturer produces the goods and delivers to the Bulk Buyer. The process is a *choreography* since there is no party that sees all messages. If all messages were sent and received by the Manufacturer, it would be an *orchestration* with the Manufacturer serving as a mediator [7].

Conflict example. This simple scenario already involves five participants who would likely blame each other in case of delays and errors. Consider the case that the Manufacturer receives the materials three days later than agreed, with eight pallets being delivered instead of ten. The Supplier might argue that this is exactly in line with what was ordered by the Middleman while the Middleman would claim the fault to be on the side of the Supplier. The situation is delicate for the Carrier since the Manufacturer refuses to accept the delivery. The Carrier is now eligible for a compensation by the Supplier or the Middleman depending on who is responsible for the fault.

2.2 Prior Research on Collaborative Business Processes

Prior research on collaborative business processes has intensively investigated different notions of compatibility between the local processes of different partners and between local processes and a global process. Such compatibility can be

achieved by design, for instance using a P2P approach [20], transformations from a global choreography [7, 22], or interaction modeling [2].

Business processes involve different trust issues (see e.g. [21] for a summary) which can be addressed in different ways. For example, [1] relaxed the assumption that the broker hosting the process engine has to be trusted: using selective encryption, data access for both the broker and the service partners can be restricted. [8] designed a trust service for cross-company collaboration based on a hybrid architecture mixing a trusted centralized control with untrusted peer-to-peer components. [6] put forward an agent-based architecture that can remove the scalability bottleneck of a centralized orchestration engine, and provides more efficiencies by executing portions of processes close to the data they operate on. In virtual organizations, [15] proposed to select partners on the basis of disclosure policies and credentials (i.e. identity attributes issued by a "Credential Authority").

Various important concepts such as conformance [19], reliability [16] and quality of services [24] have been investigated for centrally controlled business process execution. However, these works do not solve the trust issue: a collaborating party might have corrupted their historic files to their advantage. Technologies such as shared data stores provide solutions via consensus protocols to synchronize replicas [5] in a fully trusted environment. In this paper, we build our approach on blockchain technology for reasons explained next.

2.3 Blockchain Technology

Blockchain is the technology that supports Bitcoin [9]. The Bitcoin blockchain is a public ledger, which stores all transactions of the Bitcoin network. This concept has been generalized to distributed ledger systems that verify and store any transactions without coins or tokens [17]. A key feature of a blockchain-based system is that it does not rely on any central trusted authority, like traditional banking or payment systems. Instead, trust is achieved as an emergent property from the interactions between nodes within the network.

The blockchain data structure is an ordered list of blocks. *Blocks* are containers aggregating transactions. Every block is identifiable and linked to the previous block in the chain. *Transactions* are identifiable data packages that store parameters (such as monetary value in case of Bitcoin) and results of function calls in smart contracts. The integrity is ensured by cryptographic techniques. Once created, a transaction is signed with the signature of the transaction's initiator, which indicates e.g. the authorization to spend the money, create a smart contract, or pass the data parameters associated with the transactions.

If the signed transaction is properly formed, valid and complete, it is sent to a few other nodes on the blockchain network, which will further validate it and send it to their peers until it reaches every node in the network. This *flooding approach* guarantees that a valid transaction will reach all the connected nodes in the network within a few seconds. The senders do not need to trust the nodes they use to broadcast the transactions, as long as they use more than one to ensure that it propagates. The recipient nodes do not need to trust the sender either because the transaction is signed. When a transaction reaches a *mining node*, it is verified and included in a block. Blockchain networks rely on miners to aggregate transactions into blocks and append them to the blockchain. Once the transaction is confirmed by a sufficient number of blocks, it becomes a

permanent part of the ledger and is accepted as valid by all nodes.

A *smart contract* is a user-defined program executed on the blockchain network [12]. It can be used to reach agreement and solve common problems. Smart contracts can be enforced as part of transactions, and are executed across the blockchain network by all connected nodes. The blockchain platform Ethereum views smart contract as a first-class element, and offers a built-in Turing-complete scripting language for writing smart contracts, called *Solidity*. Its execution environment, the *Ethereum Virtual Machine (EVM)*, comprises all full nodes on the network and executes bytecode compiled from Solidity scripts. Trust in the correct execution of smart contracts extends directly from regular transactions, since (i) they are deployed as data in a transaction, and hence immutable; (ii) all their inputs are through transactions; and (iii) their execution is deterministic. Deployed contracts should be tested. Whether the bytecode can be trusted is a separate matter, which we discuss for our approach in Section 4.5.

3 Blockchain-based Collaborative Process Execution

In the following, we propose a blockchain-based system to address the lack-of-trust problem in collaborative business processes. A number of technical challenges arise during the adoption of blockchain for this purpose. For example, since transactions, computation, and data storage in blockchain platforms are not cost-free, not all aspects of collaborative processes should be dealt with inside smart contracts. However, smart contracts cannot call external APIs outside the blockchain environment or directly create blockchain transactions.

In the following we use the concrete languages BPMN (for process models) and Solidity (for smart contracts). Most concepts can be ported to other languages, but details will vary. Albeit Solidity is Turing-complete, there are a number of practical challenges when using it to implement processes. Among others, a contract cannot create normal transactions. Transfers of balances, updates to shared state, or messages or notifications can be produced by a smart contract – but as a consequence of the computation, not as first-class citizens of a blockchain, like transactions. Finally, since all smart contracts are distributed to all full nodes of the network, they cannot contain any confidential information, like private keys. This section presents our approach and how it addresses the challenges encountered.

3.1 Overview of the Approach

An overview of our approach is shown in Fig. 3.1. We use blockchain to facilitate the collaborative processes in either of two ways:

(i) As a *choreography monitor*, it stores the process execution status across all involved participants by observing the message exchanges. In this setting, blockchain serves as an immutable data storage to share the process execution status and create an audit trail. Smart contracts check if interactions are conforming to the choreography model. In addition, a choreography monitor can be used to manage automated payment points and escrow.

(ii) As an *active mediator* among the participants, it coordinates the collaborative process execution. This includes all the above as well as using smart

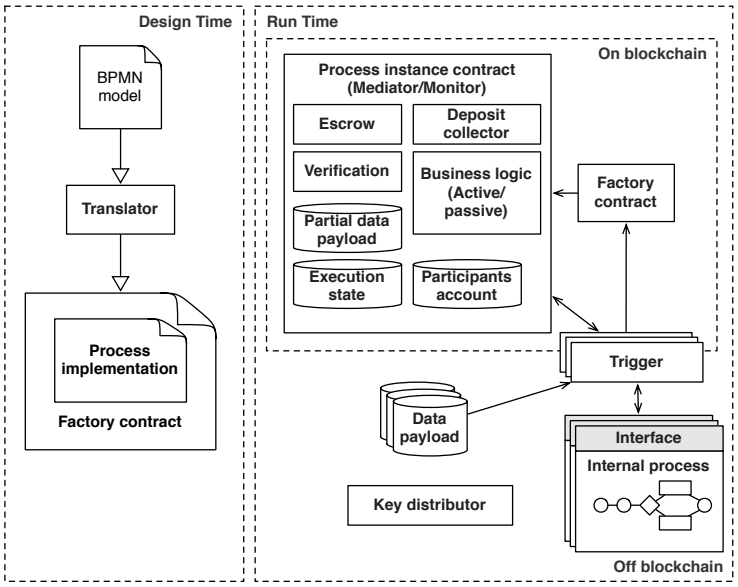

Figure 3.1: Overview of our approach

contracts to drive the process and implement data transformation or calculations.

These options are supported by the following main components:

- At design time, a **translator** derives from a process specification described in, e.g., Business Process Model and Notation (BPMN), a smart contract in a script language (such as Solidity). The generated smart contract is a factory for mediators or choreography monitors.

- For Option (i), a **Choreography monitor** or **C-Monitor** uses smart contracts *to monitor* the collaborative business processes. The C-Monitor is split into a factory and case-specific instance C-Monitors. The factory instantiates the case-specific monitors as needed, and contains the blueprint for instance C-Monitors. The C-Monitor instance tracks the interactions of a choreography instance and combines them into a consolidated view of the current state of the execution. Optionally, it can trigger automatic conditional payment from escrow, when certain points in the choreography are reached.

- For Option (ii), an active **mediator** uses a smart contract to *implement* the collaborative business processes. As with the C-Monitor, it is split between a factory and a set of instances and offers a consolidated view of the process state. In contrast to the C-Monitor, the mediator always plays an active role, receiving and sending messages according to the business logic defined in the process model. It also may transform data or execute other computations.

- **Interfaces** or **triggers** connect the process executing on blockchain and the external world. Because smart contracts cannot directly interact with the world outside the blockchain, a trigger plays the role of an organization's agent. It holds confidential information and runs on a full blockchain node,

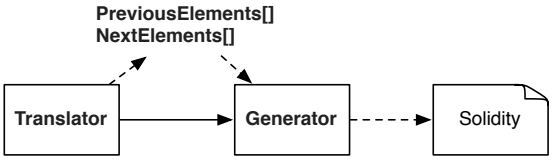

Figure 3.2: Overview of translator and generator

keeping track of the execution context and status of running business processes. The trigger calls external APIs if needed, receives API calls from external components, and updates the process state in the blockchain based on external observations. It further keeps track of data payload in API calls and keeps the data in an external database when appropriate.

By the help of these components, we achieve that (i) participants can execute collaborative processes over a network of untrusted nodes, (ii) only conforming messages advance the state of the process, (iii) payments and escrow can be coded into the process, and (iv) an immutable ledger keeps a log of all transactions, successful or not. Next, we explain the above components in more detail.

3.2 Design Time: Translator

The translator is used at design time: it takes an existing business process specification as input and generates smart contracts. These implement the C-Monitor or mediator and can be deployed and executed on the blockchain.

In a collaborative process, this functionality must be split and distributed between the smart contract and the triggers. The translator creates the artifacts in such a way that the triggers and the smart contract can collaborate directly with each other over the blockchain network.

When the translator is called, it may not be known which participants will play which roles. Therefore, the translator outputs only a **factory contract**, which in turn contains all information needed for instantiating the process. The factory contract includes the methods for instantiation and two types of artifacts: (i) an interface specification per role (e.g., buyer, manufacturer, and shipper) in a collaborative process, to be distributed to the respective triggers, and (ii) a process instance contract, which is deployed to the blockchain when the process is instantiated. The process instance contract contains the implementation of the business logic and takes the form of a C-Monitor or mediator, depending on the content of the original process specification.

The overall translation algorithm has two phases (see Figure 3.2). First, the translator parses the input process model and iterates through all its elements, where it generates two lists per element in the process model: one list of previous elements and one of next elements. Then, the translator translates each element with its respective links, generating Solidity code based on the translation rules for different types of elements as detailed below. Note that, in the current implementation, only some combinations of consecutive gateways can be connected to each other without tasks in between. The previous element list is used by the translator to determine which other elements need to be deactivated when the current element is executed; the next element list specifies which elements need to be activated after the current element is executed.

The selection methods for the two lists used by the **translator** are shown in Algorithm 1. *NextElements* of an element includes all the tasks that directly follow the element, or the outgoing edge if the target of that edge is an AND-Join. If a next element is a Split or XOR-Join gateway, the tasks / edges that connect to it are added into *NextElements* through a recursive call. *PreviousElements* of an element includes the element itself. If an XOR-Split gateway $Split_i$ precedes the current element, the tasks that follow it are added to *PreviousElements*. In the case of an AND-Join gateway, all incoming edges are added to *PreviousElements*.

Algorithm 1 Calculating PreviousElements and NextElements.

```
1:  function SelectNextElements(Element, NextElements[])
2:      for all Edge_j ∈ outgoingEdges[Element] do
3:          if Edge_j.targetElement is Task then
4:              NextElements ← Edge_j.targetElement
5:          else if Edge_j.targetElement is AND-Join gateway then
6:              NextElements ← Edge_j
7:          else if Edge_j.targetElement is Split or XOR-Join gateway then
8:              SelectNextElements(Edge_j.targetElement, NextElements[])
9:          end if
10:     end for
11: end function
12:
13: function SelectPreviousElements(Element, PrevElements[])
14:     PrevElements ← Element
15:     if Element is Task then
16:         for all Edge_i ∈ incomingEdges[Element] do
17:             if Edge_i.sourceElement is XOR-Split gateway then
18:                 SelectNextElements(Edge_i.sourceElement, PrevElements[])
19:             end if
20:         end for
21:     else if Element is AND-Join gateway then
22:         for all Edge_i ∈ incomingEdges[Element] do
23:             PrevElements ← Edge_i
24:         end for
25:     end if
26: end function
```

We extended the specification of jbpm-bpmn to support annotating payment tasks on the existing tasks. We use the JSON format to specify which account is supposed to transfer how much to the contract's account, or which account is supposed to receive how much from the contract. If the translator sees the annotation during parsing the BPMN model, it extends the corresponding task function with the logic of conditional payment.

The **generator** is based on the workflow patterns [18]. Some patterns can be directly translated, some have to be supported off-chain, and other are unnecessary in our case. Our focus is not on supporting all elements of BPMN, but we start from the 5 basic control flow patterns [18], which are among the most frequently used elements in process models [25]. We first give an overview

BPMN element	**Scope**	**Solidity code summary**
All patterns	All	On execution, deactivates itself and activates the subsequent element.
Parallel-Split	All	Executes on activation, activates *all* subsequent elements.
Parallel-Join	All	Executes on activation of *all* incoming edges.
XOR-Split	All	Executes on activation, conditionally activates all subsequent elements. If one of them is executed, it deactivates all others.
XOR-Join	All	Executes on activation of *one* incoming edge.
Choreography Task	All	Executes when the respective message is received (as blockchain transaction), and if the task is activated (message conforms with process). If conforming, the message is forwarded (as smart contract log entry); else, an alert is broadcasted.
Task: Payment	M,CME	Execution and conformance check as above. If conforming, payment into or from escrow is processed. Incoming payment is through a transaction, which has the desired effect already. Outgoing payment is sent to the account of the specified role.
Task: Data Transformation	M	Execution and conformance check as above. Mediator-internal logic on data transformation, to be handled on-chain by the mediator or off-chain by a designated trigger.

Table 3.1: Translation rule summary. During traversal of the process model, when the translator encounters a pattern (left column), it inserts code according to the right column into the smart contract code. Scope concerns which variants the pattern applies to (M: mediator; CME: C-Monitor with escrow).

of the translation rules in Table 3.1, with respect to BPMN 2.0 choreography diagram elements and their translation to Solidity. These make use of the two lists derived above, for activation / deactivation.

The process instance contract (see Algorithm 2) is generated from the translator. The process instance contract consists of a list of storage variables that represent the execution state of the process instance. To optimize the cost, we minimized the size of the data stored on chain. Two types of elements in a business process are implemented as functions in Solidity, namely, tasks and AND-Join gateways.

To start the process execution, the first task is activated in Function $Init()$. Function $Task_i()$ is invoked by the trigger to execute the corresponding task and drive the process. Function $JoinGateway_i()$ is invoked internally to enforce the control flow patterns. The storage variables representing the process execution state are manipulated and updated by every function. After the last task is executed, the variable of $TerminationActivated$ is set to be true, which terminates the process.

Algorithm 3 shows how each task $Task_i$ of a business process is implemented

Algorithm 2 solidity contract.

```
1: Bool Task_1Activated ← false
2: ...
3: Bool Task_nActivated ← false
4:
5: Bool JoinGateway_1Incoming_1Activated ← false
6: ...
7: Bool JoinGateway_1Incoming_nActivated ← false
8: Bool JoinGateway_nIncoming_1Activated ← false
9: ...
10: Bool JoinGateway_nIncoming_nActivated ← false
11:
12: Bool TerminationActivated ← false
13:
14: function INIT()
15:     Task_1Activated ← true
16: end function
17:
18: function TASK_1()
19:     ...
20: end function
21: ...
22: function TASK_n()
23:     ...
24: TerminationActivated ← true
25: end function
26:
27: function JOINGATEWAY_1()
28:     ...
29: end function
30: ...
31: function JOINGATEWAY_n()
32:     ...
33: end function
```

as a function $Task_i()$ in Solidity. The function returns a Boolean value that indicates whether the task is completed successfully. The respective "completed" variable defines the execution state of a task; it is stored in the process instance contract and manipulated by the corresponding $Task_i()$. The set of these variables defined the execution state of the process instance. Payment tasks are always performed on-chain. Computational tasks, e.g. for data transformation, could be performed on-chain or off-chain depending on the cost analysis.

$Task_i()$ performs conformance checking when receiving a message to execute Task_i. If conforming (Task_i is activated), the message is forwarded (as smart contract log entry); else, it returns false to indicate that the execution of Task_i is not succeeded and an alert is broadcasted. Once Task_i is successfully executed, all the elements of $PreviousElements$ are deactivated and all the elements of $NextElements$ are activated.

Algorithm 3 Every task is encoded in a function returning a Boolean value.

```
1: function TASK_i()
2:     PreviousElements[]
3:     NextElements[]
4:     NextJoins[]
                                              ▷ Conformance checking
5:     if Task_i Activated == false then
6:         return false
7:     end if
                                    ▷ Deactivate previous elements
8:     for all Element_m ∈ PreviousElements do
9:         Element_m Activated ← false
10:    end for
       ▷ Activate next elements, and invoke followup checks if the element is a
   AND-Join gateway
11:    for all Element_n ∈ NextElements do
12:        Element_n Activated ← true
13:        if Element_n is an incoming edge of a AND-Join gateway then
14:            NextJoins ← Element_j.targetElement
15:        end if
16:    end for
17:    for all Join_k ∈ NextJoins do
18:        Join_k()
19:    end for
20:    return true
21: end function
```

Algorithm 4 shows that a Join gateway Join_i of a business process is also implemented as a function $Join_i()$. Similarly as above, the function $Join_i()$ starts from conformance checking to make sure that the Join gateway is activated to be executed. The conformance checking is specific to the workflow patterns. For each AND-Join, it checks the condition that all the elements of $PreviousElements$ are activated. After the conformance checking, the gateway is executed, and similarly as above, all the elements of $PreviousElements$ are deactivated and all the elements of $NextElements$ are activated.

After generating the smart contracts, the translator also calculates the cost range for executing the resulting smart contract. This serves as an indication of how much crypto-coins have to be spent in order to execute process instances over the blockchain.

3.3 Runtime Environment: Executing Processes as Smart Contracts

The translator generates all artifacts needed for runtime execution. We start by describing C-Monitors, which allow passive monitoring of choreographies and optionally escrow. Active mediators can be seen as an extension of C-Monitors, and the additions are explained subsequently. The third important concept for runtime, the triggers, and the interaction between triggers and smart contracts

Algorithm 4 Every AND-Join gateway is encoded in a function.

```
1: function JOIN_i()
2:     PreviousElements[]
3:     NextElements[]
4:     NextJoins[]
5:     Bool Activated ← false
                                        ▷ Conformance checking for AND-Join
6:     if AND-Join then
7:         for all Element_m ∈ PreviousElements do
8:             if Element_m Activated == false then
9:                 return
10:            end if
11:        end for
12:    end if
                                                ▷ Deactivate previous elements
13:    for all Element_i ∈ PreviousElements do
14:        Element_i Activated ← false
15:    end for
       ▷ Activate next elements, and invoke followup checks if the element is
   an incoming edge of a AND-Join gateway
16:    for all Element_j ∈ NextElements do
17:        Element_j Activated ← true
18:        if Element_j is an incoming edge of a AND-Join gateway then
19:            NextJoins ← Element_j.targetElement
20:        end if
21:    end for
22:    for all Join_k ∈ NextJoins do
23:        Join_k()
24:    end for
25: end function
```

are covered afterwards. Finally, we describe how technical challenges like key distribution are handled.

Choreography Monitor. The first way of facilitating collaborative processes is to use a smart contract as C-Monitor, with optional escrow and conditional payment at certain points of the processes. How the private processes of participants are executed is not in scope here; however, we assume that they can make API calls (to their respective triggers) for coordination. For a new process instance, an instance contract is generated from the factory contract. Initialization includes registering participants and their public keys to roles. The C-Monitor instance contract contains variables for storing the role assignment and for the process execution status. During execution, the involved participants do not interact with each other directly. Instead, they use the monitor to exchange their input/output data payload and, by doing so, advance the state of the collaborative process. Consider the choreography in Fig. 3.3, which is another representation of the collaborative process from Fig. 2.1. All tasks are communication tasks between roles. By exchanging the messages through the C-Monitor, it can check conformance with the choreography and track the status.

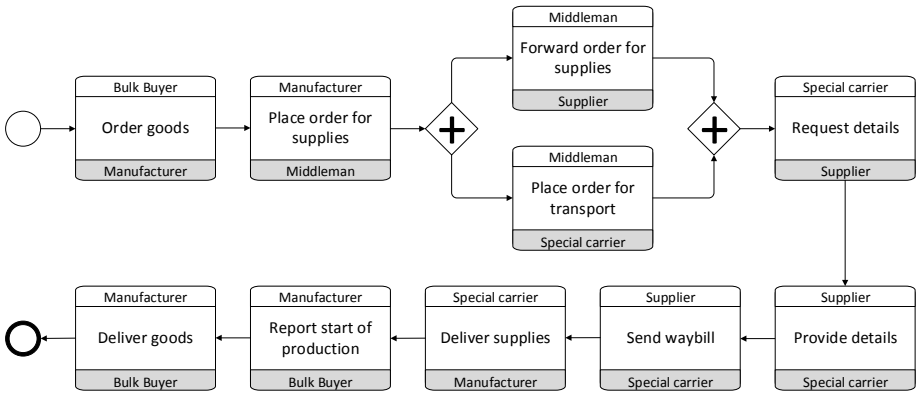

Figure 3.3: BPMN choreography diagram of the process in Fig. 2.1

While triggers and smart contracts together forward messages and update the state of the process, the state can also be inferred from the raw blockchain data. In this way, conformance checking is done implicitly by the C-Monitor, and all transactions (successful or not) are logged in the blockchain. The handling of escrow is described below.

A main design decision when using blockchain is what computation and data should be on-chain, and what should be off-chain. While the blockchain provides a trusted network in a trustless environment that can verify computational results and provide agreement on transactions' outcomes, the amount of computational power and data storage space available on the network remains limited. Besides, using the computational power and data storage space on public blockchains incurs costs. Thus, if the input/output data payload is sizable, it should be stored off-chain. In this case, the monitor stores the address of the input/output data payload. Other than the raw data or the address, we also store the hash of the data payload on the blockchain to allow verification of the integrity of the data.

Mediator. The second way of facilitating collaborative processes is to use the blockchain as an active mediator, which orchestrates the calls between the different organizations. Similar to the C-Monitor, the mediator is implemented as a smart contract, which is generated from the factory contract. It uses the same components as the C-Monitor, including registration of involved participants to roles, information specific to a process instance, and escrow. It also implements active components, among others to transform data and receive and send messages and payments.

While message and payment handling are straight-forward to achieve in smart contracts, data transformation can easily reach the point where it is not economical to implement that in a smart contract. In this case, a designated trigger can be called from the mediator, transform the data, and send a message with the output back to the mediator. Our default solution is to use the trigger of the role sending the source data for such tasks, since it has access to the data already. If data from multiple sources needs to be aggregated, the trigger of the receiving role can be used instead.

Triggers. The Blockchain is a closed environment, where the deployed smart

contracts cannot call external APIs. In our approach, a trigger (or blockchain interface) connects the participants' internal processes with the blockchain. It monitors the process execution status, logically receives messages from smart contracts and calls external APIs, or receives API calls and logically sends messages to smart contracts accordingly.

Triggers are programs running on full nodes of the blockchain network. Triggers can be distributed on multiple full nodes. In the typical setup, every participant operates its own trigger deployed on a node it controls, and the participant's systems only communicate with its own trigger. We assume that this situation is given.Alternatively, multiple participants can share one trigger if they trust each other. (In the extreme case, a single trigger could be used; but it introduces a central trusted entity and single point failure, reverting most benefits of the approach we propose.) Since the trigger is required to hold private keys for all participants on whose behalf it operates, a high degree of trust into the individual trigger is required.

When a new process instance is created, the participants register their roles and public keys. The public key corresponds to the account address of a participant. All keys and role assignments are passed to all triggers associated with the process instance, so everyone knows which role is played by whom and can verify messages accordingly. With the private key it holds, the trigger can encrypt or sign a message, allowing the contract and the other participants to verify its messages. In this fashion, it can also create payment transactions.

During the process execution, the trigger is receptive to API calls from its owner, as well as to logical messages from the process instance contract. The interaction between internal process implementations, triggers, and the process instance smart contract is shown in simplified form in Fig. 3.4. When a trigger's API is called from its owner, the trigger translates the received message into a blockchain transaction, test-calls the smart contract locally, and if that is successful sends the transaction to the instance contract. The local test call allows the trigger to check if the choreography task that expects this message is activated. If not, the local test call will return false and the trigger knows the smart contract is not in a state where the message can be sent. In turn, the trigger can alert its caller or delay the message and retry periodically. Note that, even if the local test call is successful, the real transaction can still fail, e.g., if the status has been updated between the test call and the transaction being processed. When the trigger receives a logical message from the instance contract, it updates its local state and calls an external API from the private process implementation.

Finally, the trigger takes care of sizable data payloads. For incoming API calls, it moves the data to secure storage, hashes it, and attaches a URI and the hash to the outgoing transaction. For incoming messages from the blockchain, it retrieves the data via its URI, checks if the hash matches, and sends it on to the internal process implementation.

Alternative implementation of the orchestration mediator. There are different ways to implement the orchestration mediator in terms of the distinction between on-chain and off-chain functionality. Fig. 3.5 shows how the messages flow among process participants, blockchain and trigger in different designs.

The notations in black color show the message sequence of the first design, which uses smart contract to execute the whole process, including parsing and

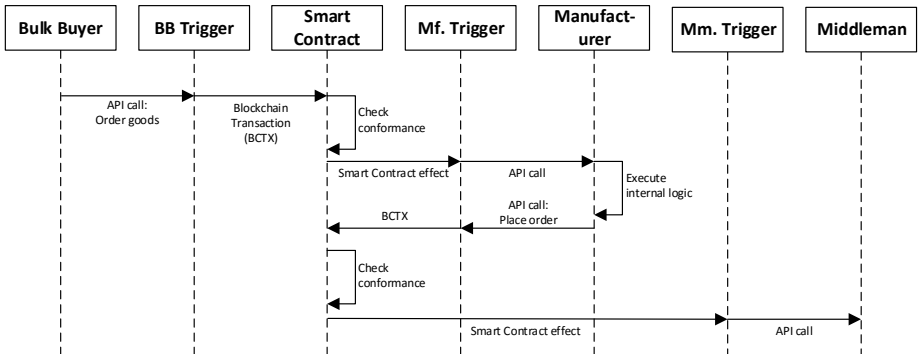

Figure 3.4: Sequence diagram for the first two tasks in Fig. 3.3

manipulating data payload, e.g., in XML or JSON format, directly in the smart contract. At the time of writing, this would likely be prohibitively expensive on public blockchains, where you have to pay for all computation done by a smart contract — but prices are constantly changing, and not all blockchains are public.

The notations in green color show the message sequence of the second design, which implements process logic in smart contract, but exchange, storage, and manipulation of data payload is handled off-chain. Due to its lower cost (only status updates are reported on the chain, only the bare process logic is executed), this option is likely more affordable on public blockchains. In this case, the triggers need to execute part of the process. Thus, at design time, the functionality of manipulating data payloads for the different activities needs to be added to the respective triggers.

The notations in blue color show the message sequence of the third design, which puts partial process execution off-chain so that part of the processes could be executed on an external execution engine. This may be applicable if two organizations trust each other, and prefer to not pay the cost of executing a part of the process that concerns only them on the blockchain.

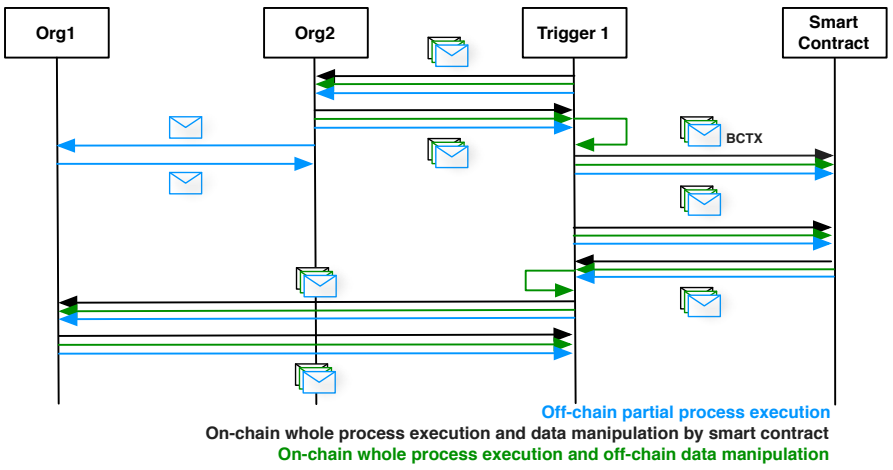

Figure 3.5: Sequence under different designs

Encryption and key distribution. All the information on the blockchain is publicly accessible to all nodes within the network. We store two types of information on blockchain, namely the process execution status and the data payload (or its URI/hash). To preserve the privacy of the involved participants, we have the option to encrypt the data payload before inserting it into the blockchain. However, the process execution status is not encrypted because the C-Monitors and mediators need to process this information. Encrypting the data payload means that mediators cannot perform data transformation at all, but can resort to the source participant's trigger for this task.

We assume the involved participants exchange their public keys with each other before a process instance is initiated by one of the involved participants. Thus, the key distribution is handled off-chain. Since participants need to find each other through off-chain mechanisms before starting a collaborative process, this typically introduces not much overhead. *Encrypting data payload for all process participants* can be achieved as follows. One participant creates a secret key for the process instance, and distributes it during initial key exchange. When a participant adds data payload to the blockchain, it first symmetrically encrypts this information using the secret key. Thus, the publicly accessible information on blockchain is encrypted, i.e., useless to anyone who has no access to the secret key. The participants involved in the process instance have the secret key and can decrypt the information. *Encrypting data payload between two process participants,* in contrast, may be desired if two participants want to exchange information privately through the process instance. For this case, the sender can asymmetrically encrypt the information using the receiver's public key; only the receiver can decrypt it with its private key.

Escrow. The C-Monitor or mediator can also work as an escrow for conditional payment at designated points. Similar to an escrow agent, e.g., in real estate transactions, the smart contract receives money from one or more parties, and only releases the money to other parties once certain criteria are met. For the receivers this has the benefit that they can observe that the money is actually there before doing work; and the sender does not have to pay upfront, trusting it will eventually receive the goods or service in return.

In the running example process, the Manufacturer (Mf) needs to pay the Middleman (Mm), Supplier (S) and Carrier (C) when it receives the goods. But S is unwilling to send the goods without some guarantees that it will get paid. Therefore, Mf puts the money in escrow, namely an account held by the process instance contract, when ordering the goods. Later, both C and Mf confirm the delivery of the goods, which triggers automatic payment from the escrow account to Mm, S, and C. The smart contract defines under what conditions the money can be transferred and how the money should be transferred. Thus, when a payment function is triggered, the smart contract automatically checks the defined conditions, and transfers the money according to the defined rules. It is, however, of high importance to specify rules that cover all possible scenarios and the respective outcomes: e.g., what shall happen with money in escrow if Mf and C disagree about the delivery of the goods or their condition? Implementing the rules in a smart contract does not prevent possible conflicts, but it allows automatic enforcement.

Gas money. The computation, data storage, and creation of smart contracts on the blockchain costs crypto-coins. That represents the cost for using the

blockchain network, since it is used to pay the miners that execute the smart contracts. Each function call is thus accompanied by cost, but contract creation is relatively much more expensive than a regular function call. For fairness, the participants in a collaborative process may want to decide on a different split of who pays how much, rather than the implicit split from the process. In our approach, the split of gas money is user-definable and can be implemented in the factory contract: it collects the money from all the involved participants and spends it on the creation of the instance contract or settles additional differences. During the execution, each participant needs to pay gas when calling a method of a smart contract.

4 Evaluation

4.1 Evaluation Method, Implementation, and Setup

The goal of our evaluation is to assess the feasibility of the approach. To this end, we implemented proof-of-concept prototypes for the translator and the trigger. The translator, written in Java, accepts BPMN 2.0 XML files, which we parse using the source code of the JBoss BPMN2 Modeller (jbpm-bpmn2 6.3.0). The translator's output are files that comply with the Solidity scripting language, version 0.2.0. Our smart contracts are running on go-ethereum 1.3.5, which is the official Golang implementation of the Ethereum protocol. The trigger is written as a Node.js web application, in JavaScript.

We picked three use case processes of different size, two from the literature and one from an industrial prototype. All three could be used directly as C-Monitor, and we extended one to cover the other options, i.e., C-Monitor with escrow and mediator. The key functionality of the blockchain is to accurately record the shared history of the choreography processes. Therefore, we derived the set of permissible execution traces for each process model, which we called the *set of conforming traces*. Furthermore, we randomly modified these traces to obtain a larger *set of not conforming traces* with the following manipulation operators: (i) add an event, (ii) remove an event, or (iii) switch the order of two events, such that the modified trace was different from all correct traces. Then we tested the ability of the smart contracts to discriminate between correct and incorrect traces. For escrow and the mediator data transformation, we ran a smaller number of experiments where we manually verified the effects.

Finally, during the above experiments we collected data that allows us to analyze important qualities. We focused particularly on cost and latency of using the blockchain in our setting, since these are the two non-functional properties that differ most from traditional approaches, such as trusted third parties. We ran experiments on a private blockchain and the public Ethereum blockchain, which allowed us to compare the effects of different options on these qualities.

4.2 Use Case Processes

For our evaluation, we used the following three processes.

1. Supply chain choreography: This process is discussed throughout this paper as a running example, see Fig. 3.3, and adapted from [3]. This process has ten tasks, two gateways and two conforming traces. From the 2

possible conforming traces, we generated 60 randomly manipulated traces. Out of these, 3 were conforming (switched order of parallel tasks) and 57 not.

2. Incident management choreography: This process stems from [11, p.18]. This process has nine tasks, six gateways and four conforming traces. We generated 120 not conforming traces. We implemented it with and without (i) a payment option and (ii) data manipulation in a mediator.

3. Insurance claim handling: This process is taken from the industrial prototype Regorous[1]. Choreographies tend to result in a simplified view of a collaborative process, as can be seen when comparing Figures 2.1 and 3.3. To test the conformance checking feature with a more complex process, we added a third use case which was originally not a choreography. This process has 13 tasks, eight gateways and nine conforming traces. We generated 17 correct and 262 not conforming traces.

4.3 Identification of Not Conforming Traces

For this part of the evaluation, we investigate if our implementation accurately identifies the not conforming traces that have been generated for each of the models. The results are shown in Table 4.1. All log traces were correctly classified. This was our expectation: any other outcome would have pointed at severe issues with our approach or implementation.

Process	**Tasks**	**Gate ways**	**Trace type**	**Traces**	**Correctness**
Supply chain process of Fig. 3.3	10	2	Conforming	5	100%
			Not conforming	57	100%
Incident management	9	6	Conforming	4	100%
			Not conforming	120	100%
Incident management with payment	9	6	Conforming	4	100%
			Not conforming	19	100%
Incident mgmt. with data transformation	9	6	Calculation	10	100%
			String manipulation	10	100%
Insurance claim	13	8	Conforming	17	100%
			Not conforming	262	100%

Table 4.1: Process use case characteristics and conformance checking results

4.4 Analysis of Cost and Latency

In this part of the evaluation, we investigate the cost and latency of involving the blockchain in the process execution, since these are the non-functional properties that are most different from solutions currently used in practice.

Cost. In our experiments on the private blockchain, we executed a total of 7923 transactions, at zero cost. On the public Ethereum blockchain, we ran

[1] http://www.regorous.com/ . A subset of the authors is involved in this project.

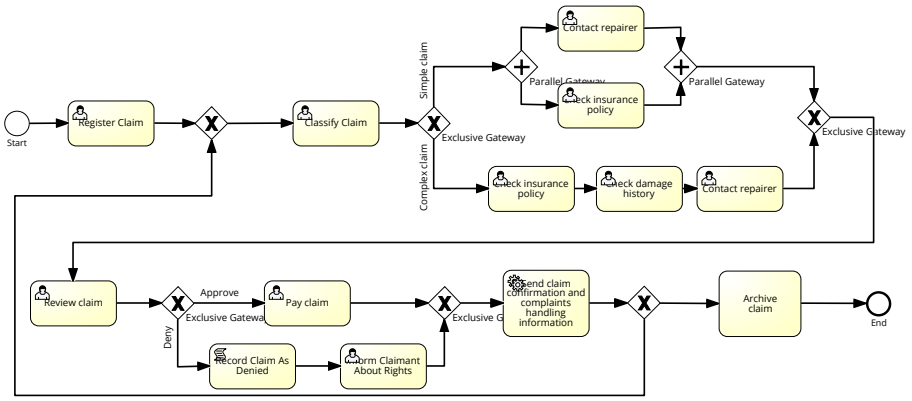

Figure 4.1: Process model: insurance claim

32 process instances with a total of 256 transactions. The deployment of the factory contract cost 0.032 Ether, and each run of the Incident Management process, with automatic payments and data transformations, cost on average 0.0347 Ether, or approx. US$ 0.40 at the time of writing. The data (transactions and contract effects) of the experiment on the public blockchain is publicly viewable from the factory contract's address, e.g. via Etherscan.[2]

Latency. We measure latency as the time taken from when the trigger receives an API call until it sends the response with conformance outcome, transaction hash, block number, etc. A test script iterates over the events in a trace and synchronously calls the trigger for each event. Therefore, the test script sends the next request very soon after receiving a response. This distorts the latency measurement to a degree, since the trigger adds the next transaction to the transaction pool just after the previous block has been mined, and it needs to wait there until mining for the block after the current one is started. Our measurements should thus be regarded as an upper bound, rather than the typical case.

In more detail, latency occurred at four points:

(i) Network latency for API calls – which is negligible over LAN and not measured in the above-described setup.

(ii) Processing by the trigger implementation.

(iii) Waiting until the transaction is added to a block: say, after block 100 has been mined the trigger submits transaction TX_1, which is added to the transaction pool. The miner is mining block 101, and will not consider new transactions until that completes. TX_1 is thus included in block 102 at the earliest.

(iv) The test script implementation: While this is not a concern in real-life situations, our test script waits until it knows TX_1 has been added to block 102 before sending the next message, and the trigger adds the next transaction. Therefore, TX_2 is added to the transaction pool shortly after

[2] https://etherscan.io/address/0x09890f52cdd5d0743c7d13abe481e705a2706384

block 102 was announced as completed – which means it will typically be added to block 104.

In summary, the most significant latency in our experiments comes from the way transactions are being inserted into the blocks. Assuming a uniform distribution of when API calls are sent to triggers, which is a fair assumption in many real-world deployments, this would on average take roughly 1.5 times the duration for a block to be mined. In our tests, due to the test script implementation, i.e. factor (iv) above, latency is measured as $1.8 - 1.99$ times that duration: the next transaction TX_1 is added to the pool at the start of the mining time for block 101 in the example, but only added to block 102. That is the reason why the latency measurements in our experiments should be seen as an upper bound.

The duration for a block to be mined comes from the complexity of the mining task, which is deliberately designed to be computationally hard. On the public Ethereum blockchain, the *target median time* between blocks at the time of writing was set to around 13s, with the actual time measured as 14.4s. This target time is controlled by the code running Ethereum[3]. The difficulty of the mining task is controlled so that it achieves a stable median time between blocks regardless of the computational power available in the network: if more power becomes available and the median time goes down, the difficulty of the mining task is automatically increased to slow the miners down. In our setup with the public Ethereum blockchain, we measured a median latency of 23.0s. A summary of the measurements is shown as *Public Ethereum* in Fig. 4.2.

In contrast, on a private or permissioned blockchain the mining speed can be controlled by changing the source code: we can make the mining task simpler. With such control, the average time between adjacent blocks was 1.38s, and the median latency measured with the test script was 2.8s. The measurements are summarized as *Private fast* in Fig. 4.2. Without this control, the mining task gets harder until it reaches a median of at least 13s. In our experiments, the median latency with the script measured at 27.4s. The measurements are summarized as *Private uncontrolled* in Fig. 4.2. For any application, this *tradeoff* needs to be considered: public blockchains offer much higher trustworthiness in return for higher cost and latency.

4.5 Discussion

Conflict resolution. Following up on the *conflict example* from Section 2.1, we discuss how conflict resolution can be implemented in our approach. Recall that there was disagreement about the amount of supplies ordered. The blockchain inherently provides an immutable audit trail, thus it is trivial to review the original order and waybill messages – the culprit can be identified through such inspection. Say, the Supplier was at fault, but the Manufacturer paid crypto-coins into escrow – how does it get its money back? The conditions for reimbursement from escrow need to be specified in the smart contract, but then they can be invoked at a later time. For instance, the participants may agree upfront that the Manufacturer gets reimbursed only if the Middleman agrees to that; then the Middleman sends a transaction to that effect, and the Manufacturer's money is transferred back to its account.

[3] https://github.com/ethereum/EIPs/blob/master/EIPS/eip-2.mediawiki

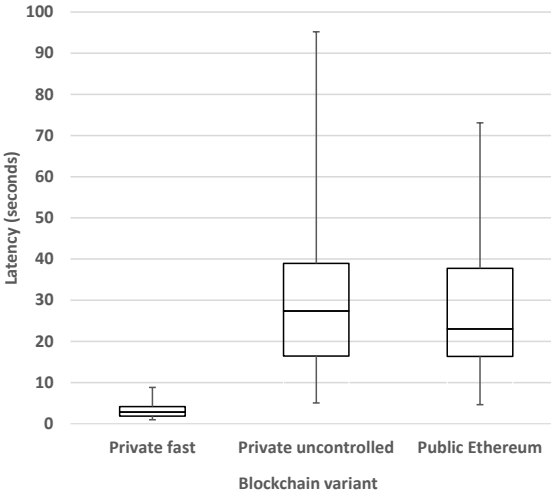

Figure 4.2: Latency in seconds, using private blockchain with / without speed modification, and public Ethereum blockchain (box plot)

Trust. Blockchain provides a trustworthy environment, without requiring trust in any single entity. In contrast, in the traditional model participants who do not trust each other need to agree on a third party which is trusted by all. Blockchain can replace this trusted third party. This is of particular interest in cases of coopetition. If multiple parties come together to achieve a joint business goal, but some of the organizations are in coopetition, it is important that the entity which executes the joint business process is neutral. Say, *Org1*, *Org2*, and *Org3* are in coopetition, but want to have a joint process to achieve some business goal. However, *Org1* would not accept *Org2* or *Org3* to control the process, and neither of those would accept *Org1*. Using our approach, the blockchain can be used, enabling trustless collaboration as it is not controlled by a single entity. Our translator allows the deployment of business processes on blockchain network without the need to manually implement the corresponding smart contract. *Trust in the deployed bytecode* for a process is established as follows: each participant has access to the process model, translates it to Solidity with our translator, and uses an agreed-upon Solidity compiler. This results in the same bytecode, and each participant can verify that the deployed bytecode has not been manipulated. Finally, the *trigger* allows for seamless integration into service-based message exchanges. However, each trigger is a fully trusted party, and by default we assume each organization hosts their own trigger.

Privacy. Public blockchains do not guarantee any data privacy: anyone can join a public blockchain network without permission, and information on the blockchain is public. Thus, for scenarios like collaborative process execution, a permissioned blockchain may be more appropriate: joining it requires explicit permission. Even with permission management, the information on blockchain is still available to all the participants of the blockchain network. While we propose a method to encrypt the data payload of messages, the process status information is publicly available. As such, if *Org1*'s competitor, *Org4*, knows which account address belongs to which participant, it can infer with whom *Org1*

is doing business and how frequently. This can be mitigated by creating a new account address for each process instance: the space of addresses is huge, and account creation trivial. However, this method prevents building a reputation, at least on the blockchain.

Off-chain data Store. For large data payloads, we propose to store only metadata with a URI on-chain, and to keep the actual payload off-chain – accessible with the URI. Due to size limits for data storage on current blockchains [14] and associated costs, this solution can be highly advantageous. There are existing solutions that provide a data layer on top of blockchains, such as Factom [14]. Distributed data storage, like IPFS, DHT (Distributed Hash Table), or AWS S3, can also be used in combination with the blockchain to build decentralized applications.

Threats to Validity. There are several limitations to our study. To start, we made some assumptions when implementing our evaluation scenario, which bear threats to validity. First, we considered a supply chain scenario in which seconds of latency are typically not an issue. We expect that scenarios in other industries, such as automatic financial trading, would have stronger requirements in terms of latency, which could limit the applicability of our technique. Second, we worked with a network of limited size. A global network might have stronger requirements in terms of minimal block-to-block latency to ensure correct replication. These threats emphasize the need to conduct further application studies in different settings. Furthermore, there are open questions regarding technology acceptance, including management perception and legal issues of using blockchain technology.

5 Conclusion

Collaborative process execution is problematic if the participants involved have a lack of trust in each other. In this paper, we propose the use of blockchain and its smart contracts to circumvent the traditional need for a centralized trusted party in a collaborative process execution. First, we devise a translator to translate process specifications into smart contracts that can be executed on a blockchain. Second, we utilize the computational infrastructure of blockchain to coordinate business processes. Third, to connect the smart contracts on blockchain with external world, we propose and implement the concept of triggers. A trigger converts API calls to blockchain transactions directed at a smart contract, and receives status updates from the contract that it converts to API calls. Triggers can thus act as a bridge between the blockchain and an organization's private process implementations. We ran a large number of experiments to demonstrate the feasibility of this approach, using a private as well as a public blockchain. While latency is low on a private, customized blockchain, the latency on the public blockchain may be considered too high for fast-paced scenarios. Additional benefits of our approach include the option to build escrow and automated payments into the process, and that the blockchain transactions from process executions form an immutable audit trail.

Acknowledgments

We thank Chao Li for integrating the trigger prototype with POD-Viz and recording the screencast video.

Bibliography

[1] B. Carminati, E. Ferrari, and N. H. Tran. Secure web service composition with untrusted broker. In *2014 IEEE ICWS*, pages 137–144. IEEE, 2014.

[2] G. Decker and M. Weske. Interaction-centric modeling of process choreographies. *Inf. Syst.*, 36(2):292–312, 2011.

[3] W. Fdhila, S. Rinderle-Ma, D. Knuplesch, and M. Reichert. Change and compliance in collaborative processes. In *IEEE SCC*, pages 162–169, 2015.

[4] B. B. Flynn, B. Huo, and X. Zhao. The impact of supply chain integration on performance: A contingency and configuration approach. *Journal of operations management*, 28(1):58–71, 2010.

[5] B. Kemme and G. Alonso. Database replication: a tale of research across communities. *Proceedings of the VLDB Endowment*, 3(1-2):5–12, 2010.

[6] G. Li, V. Muthusamy, and H.-A. Jacobsen. A distributed service-oriented architecture for business process execution. *ACM TWEB*, 4(1):2, 2010.

[7] J. Mendling and M. Hafner. From WS-CDL choreography to BPEL process orchestration. *J. Enterprise Information Management*, 21(5):525–542, 2008.

[8] M. C. Mont and L. Tomasi. A distributed service, adaptive to trust assessment, based on peer-to-peer e-records replication and storage. In *IEEE FTDCS*, 2001.

[9] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. https://bitcoin.org/bitcoin.pdf. Accessed 19/7/2015.

[10] S. Narayanan, V. Jayaraman, Y. Luo, and J. M. Swaminathan. The antecedents of process integration in business process outsourcing and its effect on firm performance. *Journal of Operations Management*, 29(1):3–16, 2011.

[11] Object Management Group. BPMN 2.0 by Example. www.omg.org/spec/BPMN/20100601/10-06-02.pdf, June 2010. Version 1.0. Accessed 10/3/2016.

[12] S. Omohundro. Cryptocurrencies, smart contracts, and artificial intelligence. *AI Matters*, 1(2):19–21, Dec. 2014.

[13] P. M. Panayides and Y. V. Lun. The impact of trust on innovativeness and supply chain performance. *Journal of Production Economics*, 122(1):35–46, 2009.

[14] P. Snow, B. Deery, J. Lu, D. Johnston, and P. Kirby. Business processes secured by immutable audit trails on the blockchain, 2014.

[15] A. Squicciarini, F. Paci, and E. Bertino. Trust establishment in the formation of virtual organizations. In *ICDE Workshops*. IEEE Computer Society, 2008.

[16] S. Subramanian, P. Thiran, N. Narendra, G. Mostéfaoui, and Z. Maamar. On the enhancement of BPEL engines for self-healing composite web services. In *Proc. SAINT Symposium*, pages 33–39, 2008.

[17] F. Tschorsch and B. Scheuermann. Bitcoin and beyond: A technical survey on decentralized digital currencies. *IACR Cryptology ePrint Archive*, 2015:464, 2015.

[18] W. van der Aalst, A. H. M. ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow Patterns. *Distributed and Parallel Databases*, 14(1):5–51, 2003.

[19] W. M. P. van der Aalst, M. Dumas, C. Ouyang, A. Rozinat, and E. Verbeek. Conformance checking of service behavior. *ACM Trans. Internet Techn.*, 8(3), 2008.

[20] W. M. P. van der Aalst and M. Weske. The P2P approach to interorganizational workflows. In *Proc. CAiSE*, pages 140–156, 2001.

[21] W. Viriyasitavat and A. Martin. In the relation of workflow and trust characteristics, and requirements in service workflows. In *Informatics Engineering and Information Science*, pages 492–506. Springer, 2011.

[22] I. Weber, J. Haller, and J. Mülle. Automated derivation of executable business processes from choreograpies in virtual organizations. *International Journal of Business Process Integration and Management (IJBPIM)*, 3(2):85–95, 2008.

[23] I. Weber, X. Xu, R. Riveret, G. Governatori, A. Ponomarev, and J. Mendling. Untrusted business process monitoring and execution using blockchain. In *BPM'16: International Conference on Business Process Management*, Rio de Janeiro, Brazil, Sept. 2016.

[24] L. Zeng, B. Benatallah, A. Ngu, M. Dumas, J. Kalagnanam, and H. Chang. Qos-aware middleware for web services composition. *IEEE TSE*, 30(5):311–327, 2004.

[25] M. zur Muehlen and J. Recker. How Much Language is Enough? Theoretical and Practical Use of the Business Process Modeling Notation. In *Proc. CAiSE*, 2008.