SN Computer Science manuscript No.
(will be inserted by the editor)

Workflow Management on Proof-of-Work Blockchains — Implications and Recommendations

Joerg Evermann · Henry Kim

Abstract Blockchain technology, originally popularized by cryptocurrencies, has been proposed as an infrastructure technology with applications in many areas of business management. Blockchains provide an immutable record of transactions, which makes them useful in situations where actors must cooperate but may not fully trust each other. In this paper we examine the use of proof-of-work blockchains for executing inter-organizational workflows. We discuss architectural options and describe two prototype implementations of a blockchain-based workflow management system (WfMS), highlighting differences to traditional WfMS. Our main contribution is the identification of potential problems raised by proof-of-work blockchain infrastructure and recommendations to address them.

Keywords Blockchain · proof-of-work · workflow management · inter-organizational workflow · distributed workflow · collaboration

Declarations

– The authors received no funding in support of this research.
– The authors have no conflicts of interests or competing interests with respect to this research.
– The authors will make all source code publically available after acceptance.

An earlier version of this paper can accessed at http://arxiv.org/abs/1904.01004.

J. Evermann
Memorial University of Newfoundland, St. John's, Canada
E-mail: jevermann@mun.ca
`https://joerg.evermann.ca`

H. Kim
York University, Toronto, Canada

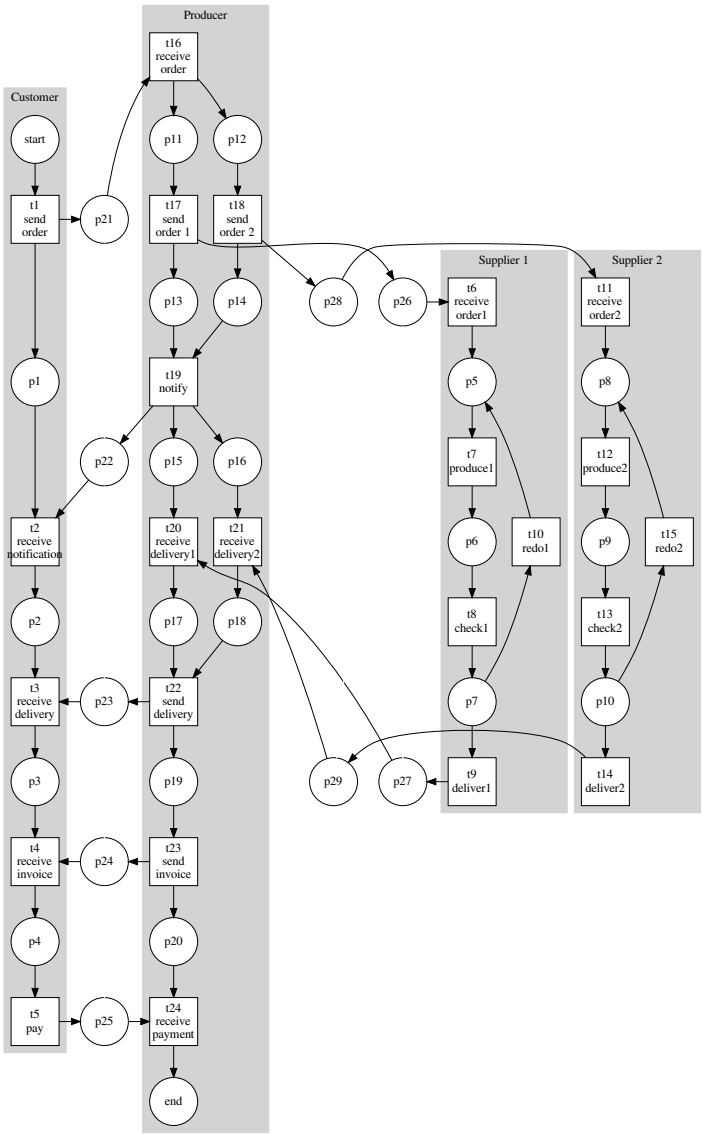

Fig. 1 Inter-organizational process modelled as Petri net, adapted from van der Aalst (2000)

1 Introduction

Workflow management (WfM) has traditionally focused on intra-enterprise applications. Despite its many challenges, inter-organizational WfM has seen less research attention. Fig. 1 shows an example of an inter-organizational process, adapted from van der Aalst (2000). The four shaded areas represent four

different participants in this process, boxes represent activities in the process to be performed by different participants. In this process, the customer orders goods from a producer, who in turn relies on two suppliers to provide required parts. The process begins with the customer sending the order to the producer and ends when the producer has received payment. Inter-organizational processes may include stakeholders that are in adversarial relationships with each other, but that nonetheless have to jointly complete process instances. In such situations, trust in the current state of a process instance and the correct execution of activities may be lacking. Blockchain technology can help by providing a trusted, distributed, workflow execution infrastructure.

A blockchain cryptographically signs a series of blocks that contain transactions, providing an immutable record. In a distributed blockchain, actors form a peer-to-peer (P2P) network to independently validate transactions and add them to the local replicas of the block chain. In inter-organizational workflow management, actors must agree on the state of work as this determines the set of valid next activities in the process. Thus, it is natural to use blockchain transactions to record either workflow activities or workflow states.

Workflow management systems (WfMS) are designed to execute business processes. They can be implemented in different ways on blockchain infrastructure. In contrast to earlier work, we do not use smart contracts to implement workflow engines on a blockchain. *While smart contracts have great potential for workflow management, they are not the only way to integrate blockchain technology with WfMS. To conclude that the problem is solved is premature at this time, given that the issue has only very recently received research attention (Mendling et al., 2018), and in light of the issues we identify in this paper. Given the extensive investment in WfMS by researchers and practitioners, it is worthwhile to investigate how existing WfMS can be implemented on blockchain infrastructure without having to re-implement them in smart contracts.*

In this paper, we show that generic or existing workflow engines can be readily adapted to fit onto a proof-of-work blockchain infrastructure and that smart contracts are not required. We investigate and propose standard interfaces between blockchain infrastructure and workflow engines. We describe two research prototype WfMS that provide proof-of-concept implementations[1] of our proposed architecture. The distributed nature of a blockchain and the nature of the proof-of-work consensus process raises challenges that are not seen in centralized WfMS. Our research prototypes have helped us to identify these challenges and to offer recommendations for future blockchain-based WfMS.

Contribution: While the prototype implementations are important demonstrations of feasibility, they are intended to be research tools only. Our main contribution is in the lessons learned from their implementation and our recommendations for proof-of-work blockchain-based WfMS. Specifically, in this paper we:

[1] Source code available from the corresponding author's website at `https://will.be.added.after.review`

1. Demonstrate the feasibility of an alternative to smart contracts for blockchain-based WfMS, including the development of generic interfaces between architecture components, and
2. Identify challenges, implications, and recommendations arising from the use of proof-of-work blockchains for workflow management. These are the same for smart contract-based architectures and for architectures not based on smart contracts, as they stem from the properties of the proof-of-work consensus mechanism.

The remainder of the paper is structured as follows. Sec. 2 introduces workflow management systems and workflow nets. Sec. 3 reviews related work on distributed, inter-organizational, and blockchain-based workflow management. Sec. 4 provides a primer on proof-of-work distributed blockchains. Sec. 5 presents the main principles of our approach and discusses validity guarantees. Next, Sec. 6 presents our prototype implementations. Implications of using blockchain technology for workflow execution are discussed in detail in Sec. 7. We conclude with recommendations for addressing the identified challenges, a comparison of architectures, and an outlook to future work (Sec. 8).

2 Workflow Management Systems and Workflow Nets

Workflow management systems (WfMS) manage the flow of work within and between organizations based on models of a business process, like the one shown in Fig. 1. Process models for execution by a WfMS are called workflow models. Whereas a workflow model specifies a process in general, the execution of a workflow model for a particular case is called a workflow instance. For example, Fig. 1 specifies an ordering process in general; handling a particular order, e.g. order number 123, is done by a specific instance of the process. A workflow activity for a case is called an activity instance or work item, e.g. activity "Send Order" for order number 123.

The core of a WfMS is the workflow engine which maintains a set of defined workflow models, a set of running workflow instances (cases), and the execution state of each case, i.e. which activities are being or have been completed. It detects deadlocked and finished cases and removes them from its set of running cases. Based on the workflow model and execution state, the workflow engine identifies for each workflow instance the next activities that must be completed and creates work items for manual or automatic execution, as specified by the workflow designer. Work items are assigned to resources (human or other), typically using role- or capability-based allocation mechanisms, and are managed in worklists. Additionally, the workflow engine maintains case information, i.e. information that is generated or required by work items. Case information for the example process in Fig. 1 may consist of purchase orders, production orders, shipping notices, invoices, and payment notices. The workflow engine may be supported by, or include, services for managing organizational data such as resources and their roles and capabilities, for managing

worklists, for providing user interfaces for interacting with the work list and with manual activities, for digital document storage, etc.

For the workflow engine to identify the next activities based on the execution state of a case, the workflow model must have a well-defined semantics. In this paper we use workflow models that are based on workflow nets, a special type of Petri net (van der Aalst, 1998). A Petri net PN is defined as a tuple $PN = (P, T, F, M)$ where P is a set of places $p \in P$, T is a set of transitions $t \in T$ such that $P \cap T = \emptyset$. $F \subseteq (P \times T) \cup (T \times P)$ is the flow relation. The preset of a transition t is $\bullet t = \{p_i | (p_i, t) \in F\}$. The postset of a transition t is $t\bullet = \{p_i | (t, p_i) \in F\}$. The preset of a place p is $\bullet p = \{t_i | (t_i, p) \in F\}$. The postset of a transition p is $p\bullet = \{t_i | (p, t_i) \in F\}$. A marking M is a mapping of places into the non-negative integers $M : P \rightarrow \mathbb{N}_0$. A transition t is enabled iff $\forall p_i \in \bullet t : M(p_i) \geq 1$. A transition t can fire iff it is enabled. Firing a transition t changes the marking M_{pre} to the marking M_{post} such that $\forall p_i \in \bullet t : M_{post}(p_i) = M_{pre}(p_i) - 1$ and $\forall p_i \in t\bullet : M_{post}(p_i) = M_{pre}(p_i) + 1$. A workflow net is a Petri net iff there exists exactly one place p_{source} such that $\bullet p_{source} = \emptyset$ and there exists exactly one place p_{sink} such that $p_{sink}\bullet = \emptyset$ and there exists an initial marking $M_0(p) = 1$ iff $p = p_{source}$ and $M_0(p) = 0$ otherwise.

The example model in Fig. 1 is a workflow net. It shows places as circles, labelled p_1 through p_{29} together with p_{source} labelled "start" and p_{sink} labelled "end". The model shows transitions as squares, labelled t_1 through t_{24}. Each transition specifies a workflow activity, e.g. the activity "Send Order" in Fig. 1. The execution states of workflow instances are described by workflow net markings. When a new workflow instance is created, it is assigned an initial marking M_0. A work item is created when a transition is first enabled in the workflow net for a case. When a work item is completed, the marking for the case is updated by firing the associated transition. The workflow engine keeps track of the markings of all running instances.

While we use Petri nets for the workflow models in this paper, any other modeling language with a sufficiently well-defined execution semantics could be used; the contributions, recommendations, and conclusions of this paper are not specific to Petri nets.

3 Related Work

A blockchain-based WfMS can be viewed as a type of distributed, replicated, inter-organizational WfMS. This section reviews prior research on distributed and replicated WfMS, inter-organizational WfMS, and the state-of-the art in blockchain-based WfMS.

3.1 Distributed Workflow Management

Distributed WfMS have seen research interest in the late 1990s and early 2000s. With the advent of client-server technology, distributed object-oriented stan-

dards such as the Common Object Request Broker Architecture (CORBA), and the beginnings of P2P networking, researchers identified ways to use these infrastructure technologies to address technical issues such as fault tolerance, redundancy, and scalability through distribution and replication.

The Exotica/FlowMark system by IBM (Alonso et al., 1995) focuses on persistent message passing between nodes when the process can be partitioned onto different workflow nodes. In the Ready system (Eder and Panagos, 1999), independent WfMS can subscribe to a shared event-publishing system. METEOR2 coordinates independent workflow systems using distributed workflow schedulers (Das et al., 1997; Miller et al., 1998). Workflow evolution in distributed systems has been studied in the ADEPT system (Reichert et al., 2003; Reichert and Bauer, 2007) while P2P network technology has been used to implement distributed "web workflow peers" that execute workflows controlled by a central administration peer (Fakas and Karakostas, 2004). The SwinDeW system (Yan et al., 2006) is another approach based on P2P technology. Efficiency of network communication has been the focus of Bauer and Dadam (1997), who develop optimal algorithms for case transfer of sub-workflows to distributed servers. A load-balancing approach by Jin et al. (2001) uses a central decision making component to distribute complete workflow instances across multiple WfMS. The event-based distributed system EVE (Geppert and Tombros, 1998) relies on synchronized clocks to distribute workflow activities to participating service execution nodes. Based on partitioning of state-charts and incremental synchronization of distributed workflow engines, the Mentor project (Muth et al., 1998) developed algorithms for optimal communication and message exchange among distributed WfMS. The Metuflow system (Dogac et al., 1998) uses transaction semantics to determine the proper sequence of activities in a distributed system that is built on a reliable message passing infrastructure and CORBA message exchange. CORBA also forms the infrastructure for an approach that uses a common monitor and scheduler to coordinate multiple "task managers" that can independently execute workflow activities (Miller et al., 1996). The Wasa2 system (Vossen and Weske, 1999) also implements a CORBA-based infrastructure of services to manage business and workflow objects. Focusing on performance and availability, continuous time Markov chains are used to derive load models and availability models for distributed WfMS (Gillmann et al., 2000). Also focusing on performance and load management, another approach employs dynamic server assignment where activities are assigned to workflow servers at runtime instead of design time (Bauer and Dadam, 2000).

Much of the work discussed above assumes a central coordination or decision-making authority. Key assumptions are either central coordination with decentralized execution of specific workflow activities, or limited case transfer to a typically homogeneous set of WfMS. The aims are mostly technical, with a focus on infrastructure suitability. Blockchain technology may be seen as another distributed infrastructure technology but it differs in key aspects from earlier technology:

1. Blockchains replicate all information to all nodes. Selective replication to optimize or minimize communication requirements is eschewed in the blockchain context as it runs counter to independent validation.
2. As all actors share a consensus view of the workflow state on the blockchain, special decision-making or control nodes are unnecessary.
3. Blockchains provide trust by providing a tamper-resistant record. Hence, building control or trust mechanisms on top of the distributed infrastructure is unnecessary.
4. The proof-of-work consensus method used in most blockchains takes an eventual-consistency approach and accepts latency when establishing consistency, which significantly relaxes the technical requirements on the infrastructure in terms of performance, security, and reliability.

3.2 Inter-organizational Workflow Management

Multiple organizations can collaborate on a single process instance in different ways, such as capacity-sharing, chained execution, subcontracting, case transfer, and loosely-coupled workflows (van der Aalst, 1999). Blockchain technology can be used to implement all of these collaboration types but may be best suited for the case-transfer collaboration, where all actors share a process definition and each actor performs different activities for a case. Much of the earlier work on distributed workflows (Sec. 3.1), and all of the blockchain-based WfMS discussed below (Sec. 3.3), assumes this type of collaboration.

The public-to-private approach considers a public workflow definition as a contract between participating actors (van der Aalst, 2002; van der Aalst and Weske, 2001; van der Aalst, 2003). Actors can provide private implementations for their parts of a process that must be compatible with the public contract. Compatibility is defined in terms of projection inheritance: The private workflows must inherit the public behavior but may offer specific implementations of this behavior. Public and private workflows are also the foundation for an architecture focusing on flexibility and respect for privacy, where details of local processes need not be publically visible (Chebbi et al., 2006). Inspired by service-oriented architecture (SOA) principles, this approach includes workflow identification and advertisement on a public registry, workflow interconnection governed by contracts ("cooperation policies") and monitoring using a trusted third party. There is no pre-defined global process model that is partitioned. Instead, the complete process model is dynamically assembled from advertised process interfaces that describe views on hidden, private processes. In the Crossflow project (Grefen et al., 2000), a process specification forms the contract for interaction among service providers. The technical architecture consists of independent WfMS coordinated by a central contract manager. The contract manager also monitors quality-of-service guarantees. Another use of contracts (Weigand and van den Heuvel, 2002) views them as "glue to link inter-organizational workflows" and provides a formal language for business communication. Workflows are managed locally and coordinated

among different actors by a central "contract object" using messages specified in the contract. Based on P2P networks, Atluri et al. (2007) describe a method to successively partition a complete process model. Each organization receives a process model whose initial activities are assigned to that organization. The organization executes its own activities, then partitions the remainder of the process for the successive organizations and passes on those partitions. A central mechanism is only required to initiate each case by identifying the first organization(s), and to accept the final results from the last organization(s).

Blockchain technology differs from these inter-organizational approaches:

1. Each organization acts independently. Executing invalid activities simply leads to transactions that will not be validated by peers and not become part of the consensus blockchain. Trusted third parties for contract monitoring or enforcement are not required.
2. Blockchain infrastructure makes all transactions publically visible. However, aspects of a workflow may be implemented by each organization privately and the notion of projection inheritance remains useful for this.

3.3 Blockchain-based Workflow Management

Blockchain-based workflow execution has only recently received research attention (Mendling et al., 2018). Existing work has focused exclusively on the use of "smart contracts" to coordinate workflow activities among participants. A smart contract is a software application that is recorded on the blockchain, "listens" for transactions sent to it, and executes application logic upon receipt of a transaction. It can itself generate messages that can be observed by participating organizations.

Driven by a financial institution, a prototype implementation using smart contracts on the Ethereum blockchain offers digital document flow for trading partners in the import/export domain (Fridgen et al., 2018). The project demonstrates significantly lowered process cost, increased transparency, and increased trust among trading partners. A project in the real-estate domain, also using the Ethereum blockchain and smart contracts, concludes that the lack of a central agency makes it more difficult for regulators to enforce obligations and responsibilities of trading partners (Hukkinen et al., 2017).

The blockchain-based WfMS by Härer (2018) uses workflow models as contracts between collaborators. The system allows distributed, versioned modelling of private and public workflows, consensus building on versions to be instantiated, and tracking of instance states on the blockchain. The blockchain provides integrity assurance for models and instance states.

Another implementation of a blockchain-based WfMS uses smart contracts on Ethereum in two ways (Weber et al., 2016). As a choreography monitor, the smart contract on the blockchain merely monitors execution status and validity of workflow messages against a process model. As an active mediator, the smart contract additionally drives the process by sending and receiving messages according to the process model. Models defined in the Business Process

Modelling Notation (BPMN) models are translated into the Solidity contract language. Peers monitor the blockchain for relevant messages from the contract and create messages to the contract. The system checks the acceptability of a response message by running it against a local copy of the contract before publishing it to the blockchain. Transaction cost and latency are recognized as important considerations in the evaluation of the approach. A comparison between the Ethereum blockchain and the Amazon Simple Workflow Service shows that blockchain costs are two orders of magnitude higher than those of a traditional infrastructure (Rimba et al., 2017). Recognizing that optimizing the space requirements for smart contracts is important, BPMN models can be translated to Petri Nets, for which minimizing algorithms are available, which are then compiled into smart contracts to achieve up to 25% reduction in transaction cost while significantly improving throughput time (García-Bañuelos et al., 2017). Building on lessons learned from Weber et al. (2016), Caterpillar is an open-source blockchain-based WfMS (López-Pintado et al., 2017). Developed in Node.js and using the Solidity compiler *solc* and Ethereum client *geth*, it provides a distributed execution environment for BPMN-based process models. Lorikeet is a similar system (Ciccio et al., 2019), also based on BPMN models that are translated to smart contracts for the Ethereum chain.

Prybila et al. (2020) use the Bitcoin blockchain to implement secure and verifiable passing of process control between participants using a token-based approach. This re-purposing of Bitcoin for choreography verification is limited by the size of Bitcoin transactions, which constrains the size of the processes and number of cases that can be monitored. The approach relies on off-chain interactions between participants to construct process handover transactions. Prybila et al. (2020) provide extensive performance evaluations, but the use of the public Bitcoin blockchain means limits to block interarrival rates and block sizes, making their approach suitable for long-running processes, but less so for fast processes. Like Prybila et al. (2020), our approach makes available the data and execution state on the blockchain to all participants. However, we store task completion rather than process handover on the blockchain and do not require explicit acceptance of control.

Modelling of blockchain-based processes requires blockchain-specific modelling constructs. BlockME provides an extension to BPMN for describing transaction state changes (Falazi et al., 2019a). It also implements an interface between blockchains and the workflow engine, allowing workflow tasks to track transaction status and submit new transactions. BlockME2 adds a measure for degree of confirmation in different blockchains and access to smart contracts (Falazi et al., 2019b). Noting that ownership of process control and visibility of data are changed when processes are executed on blockchain infrastructure, Ladleif et al. (2019) extend BPMN choreography diagrams with data objects to represent public information on the blockchain, with sub-choreographies to limit data visibility, with smart-contract controlled gateways, and with transaction-driven semantics. Their implementation can generate smart contracts for the Ethereum blockchain.

Our work differs from prior work in the following aspects:

1. Our work does not use smart contracts to implement workflow engines for specific workflow models.
2. Our work focuses on the use of off-chain, standard workflow engines and on the interfaces to the blockchain infrastructure.
3. Our work does not re-purpose existing single-purpose blockchains.
4. We are not concerned with conceptual modeling of workflows, but focus on their execution.

4 Proof-of-Work Blockchains

This section describes blockchains that use a proof-of-work consensus mechanism, as implemented in the Bitcoin cryptocurrency and the popular Ethereum blockchain. They are the most common types of blockchains and prior work in blockchain-based WfMS (Sec. 3.3) builds exclusively on such chains. However, proof-of-work is but one way to achieve a consensus ordering of blocks in a distributed system when nodes may exhibit Byzantine faults, i.e. nodes may be unreliable, unavailable, unresponsive, or malicious. Another approach, used in the Hyperledger Fabric (Androulaki et al., 2017; Sousa et al., 2018) blockchain, is based on provably live and correct ordering protocols which can be traced back to a practical method for achieving byzantine fault tolerance (PBFT) (Castro and Liskov, 2002). These kinds of ordering protocols are faster than proof-of-work and provide finality of consensus on the order of blocks. On the other hand, they are not as scalable as proof-of-work blockchains due to their communications demands (Vukolić, 2015).

A blockchain consists of blocks of transactions (Fig. 2), which can contain any kind of content. Each block also contains the hash of the content of the previous block in the chain. Hence, altering the content of a block requires changing all following blocks in the chain. For example, a change to transaction Tx12 in block 1 in Fig. 2 results in a different hash for block 1. Hence, block 2's hash needs to be recalculated, and the same for block 3 and block 4.

In a distributed blockchain, new blocks and transactions are distributed among peers. Each peer maintains a pool of transactions to be included in future blocks. New transactions to be added to this pool are independently validated by each peer, i.e. it is ensured that they are logically allowed. In the Bitcoin chain this involves ensuring that transaction inputs reference unspent transaction outputs; in the workflow context this may mean that executing a workflow activity is permitted in the current state of a process instance.

4.1 Mining

With sufficient hashing power, it becomes possible for peers to recompute earlier blocks in the chain faster than new blocks are added. Hence, they are able to "alter history" as recorded on the chain. To prevent such tampering, block hashing must be made difficult or expensive. This is typically done by

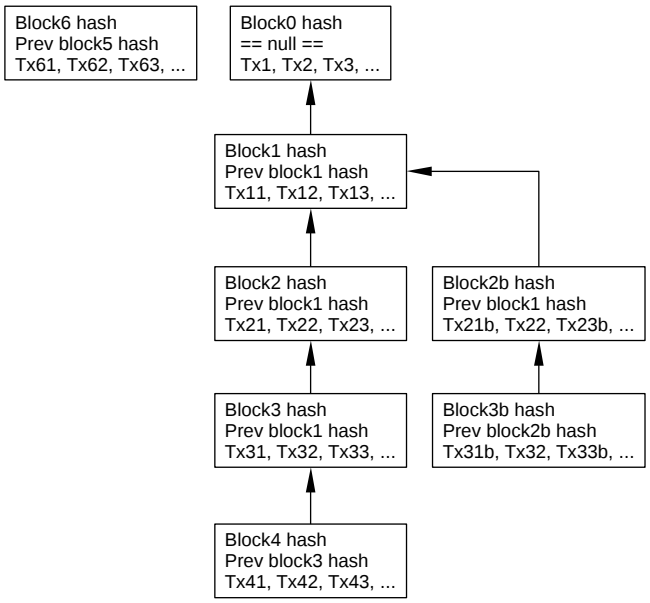

Fig. 2 Example blockchain with transactions, orphan blocks and side-branch.

requiring block hashes to have a certain number of leading zeros, the "block difficulty". Adding arbitrary content (a "nonce") to the block and repeatedly varying this nonce until a suitable hash is found is known as *proof-of-work mining*. Once a new block has been mined, it is published to all peers, and independently validated by each peer before acceptance.

4.2 Chain Reorganization

Depending on network speed and topology, new blocks and transactions arrive at peers in different order and at different times. Hence, each peer may have a different set of blocks and transactions, and hence may also mine different blocks. For example, Fig. 2 shows blocks 0–4 that reference each other through their block hashes. At the same time, this peer also possesses block 2b, possibly mined by a peer that was in possession of a different set of transactions, and followed by block 3b. Each peer considers the branch with the most mining work (typically its longest branch) as the current *main branch*. Each peer mines new blocks on top of what it considers the head of the current main branch. Side branches occur when different peers mine different blocks based on the same main branch. These may contain different transactions, as in Fig. 2, the same transactions in different order, or just a different value for the nonce. Importantly, transactions in a side branch are not considered as valid.

When a side branch becomes longer than the current main branch, the chain undergoes a *reorganization*. For example, in Fig. 2, assume that block

3b is considered the head of the current main branch and block 3 is the head of a side branch. As block 4 arrives at this peer, block 4 is now the head of the new main branch. As a result, all transactions in blocks 2 and 3, as well as those in the new block 4, must now be validated. At the same time, transactions in blocks 2b and 3b that are not in the new main branch are considered invalid and are added back to the transaction pool to be mined again. In our example, these are transactions 21b, 23b, 31b, and 33b as transactions 22 and 32 are also contained in blocks 2 and 3. The invalidated transactions will not necessarily be included in later blocks, as they may logically contradict transactions in the main branch.

4.3 Transaction Lifecycle

A transaction is said to be *submitted* when it is in the transaction pool waiting to be mined, *mined* once it is in the head block of the chain, and *confirmed*, i.e. sufficiently certain to be acted upon, when it has an agreed upon *confirmation depth*. For example, in Fig. 2, transactions Tx21, Tx22, Tx23, Tx11, Tx12, Tx13, Tx1, Tx2, and Tx3 are considered confirmed at confirmation depth of two, as there are two or more mined successor blocks in the main chain. Transactions in orphan blocks or side branches are never considered confirmed. The probability that a transaction is invalidated during chain reorganization decreases with as more blocks are mined of top of it However, even confirmed transactions can return to the *submitted* state during a chain reorganization.

In summary, the main features of proof-of-work consensus are delayed consensus that introduces a confirmation latency and non-finality of consensus. These features have significant implications for applications like WfMS built on proof-of-work blockchains.

5 Architecture, Principles, and Validity

Our architecture is for an application-specific blockchain that couples the notion of blockchain transaction validity with the permissibility of a workflow transaction. A workflow transaction may indicate completion of a work item, launching of a new case, etc. This is similar to the way in which Bitcoin, also an application-specific blockchain, couples transaction validity to the permissibility of bitcoin spending by examining unspent transaction outputs (UTXO) when validating a transaction. Hence, blockchain nodes require access to a workflow engine to validate workflow transactions. While this requirement admits many different architectural designs, we have opted for the simplest one: each blockchain node has a local workflow engine. While a more general $n : m$ relationship between blockchain nodes and workflow engines may be more flexible in deployments, it would not allow fully independent transaction validation by all process participants. Our architecture can be extended to include other types of transactions for other applications, as long as every node has access to trusted validation services for such transactions.

In our architecture, a workflow transaction originates from a workflow engine. It is passed to the workflow engine's local transaction service, which validates the transaction locally, and, if valid, publishes it on the P2P network. When a node receives a new workflow transaction, it validates the transaction locally before accepting it. When a node receives a new block, the block and all of its transactions are validated locally before being accepted. When a block is accepted, it is passed to the local workflow engine for it to update its workflow state based on each of the block's transactions. It is important to differentiate between the concepts of *"performing"* a work item in the sense of a human user or an external software application completing a task, and *"executing"* the transaction indicating completion of the work item, i.e. recording its performance and updating the resulting workflow instance state on the blockchain. While work items are *performed* on only one node, the workflow engines on all nodes *execute* the corresponding workflow transaction and thereby maintain a common workflow state.

An alternative design is to use a weaker validity criterion at the blockchain layer, i.e. to accept all transactions, and let the workflow engine filter out invalid ones when they are passed to it for execution. This approach is used by smart contract based workflow management on generic blockchains like Ethereum. There, transaction validity is independent of application-specific semantics of the validity of the action recorded in the transaction. The validity of a transaction carrying a call to a smart contract method is determined solely by the correct call parameters and sufficient "gas", not by the application logic of the smart contract itself. The contract may decide to simply not update its state when it receives a transaction representing a non-permissible workflow action. Whereas our architecture rejects non-permissible workflow transactions before they are encoded on the blockchain, the smart-contract architecture encodes them on the blockchain, together with the resulting, possibly unchanged, smart contract state.

In summary, both types of architectures make the same validity guarantees for workflow transactions, and execute only and all permissible workflow transactions on all correctly operating nodes. As long as a majority of peers agrees on what constitutes validity, that set of peers will arrive at a consensus of the blockchain and workflow state.

6 Prototype Implementations

This section presents two implementations of our architecture described in Sec. 5. These research prototypes have allowed us to explore implications of using a blockchain infrastructure for WfMS and to identify possible design choices. For ease of development, they are developed in Java.

Our architecture has three layers. The network layer forms a private P2P infrastructure with a certificate authority that issues public/private key pairs to participating actors. To keep our prototype simple, actors are identified by their internet address, rather than their public keys. However, an address

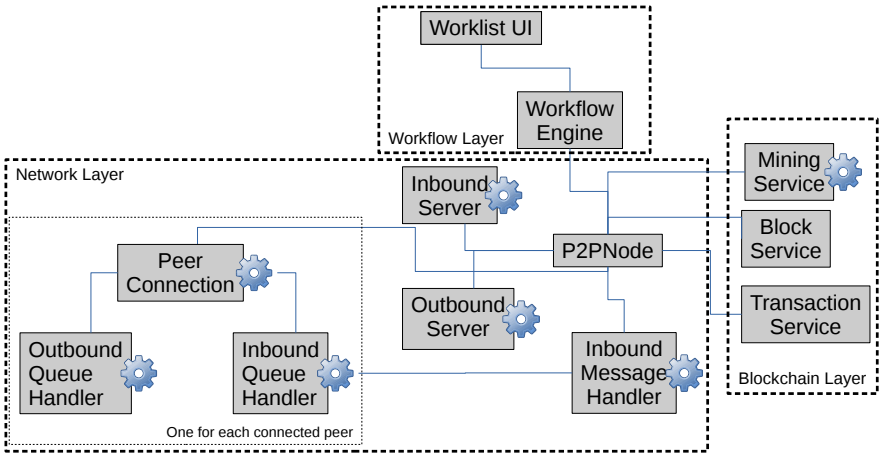

Fig. 3 Components of the prototype implementation, grouped by layer. Some components run as separate threads as indicated.

BlockRequest	Requests a block with a specific hash from a peer
BlockSend	Sends a block to one or more peers
PeersRequest	Requests a list of known peers from a peer
PeersSend	Sends a list of known peers to another peer
TransactionSend	Sends a transaction to other peers
TransactionPoolRequest	Requests the current transaction pool from a peer
TransactionPoolSend	Sends the current transaction pool to a peer
BlockchainRequest	Requests the blockchain, beginning at a certain hash from a peer
BlockChainSend	Sends the blockchain beginning at a particular hash to a peer

Table 1 Message types

resolution layer can easily be added. The P2P layer is implemented using Java sockets and serialization. As indicated in Fig. 3, each node has an outbound server that establishes connections to other peers, and an inbound server that accepts and verifies connection requests. Each connection is served by a peer-connection thread, which in turn uses inbound and outbound queue handler threads to receive and send messages. Incoming messages are submitted to the inbound message handler which passes them to the appropriate service. Messages are cryptographically signed and verified upon receipt to prevent impersonation of actors and provide non-repudiation. Table 1 describes the different message types. The P2P protocol is loosely based on that used by Bitcoin.

The blockchain layer, comprising the transaction service, block service, and mining service, is implemented on top of the P2P layer. The transaction service manages the pool of pending transactions, which are created by the workflow layer or received from the inbound message handler, and are validated upon receipt. The block service receives blocks from the mining service or the inbound message handler, validates them, and adds valid blocks to the

blockchain. It manages orphan blocks, side chains and chain reorganization. For simplicity, our prototypes use fixed-difficulty mining.

The workflow layer, comprised of the workflow engine and the worklist handler with its user interface, is implemented on top of the blockchain layer. Each activity in a workflow model is associated with a single participating node. Fig. 1 indicates this association by the shaded areas. The partitioning of the process to different nodes only signals the workflow engine whether a work item is to be handled on the local node. The process designer can provide further role information, and each node can implement its local work item allocation using that role information and local organizational information. The process designer can also describe external method calls, which are executed by the workflow engine for automated activities.

The data perspective is designed as a key–value store. We admit only simple Java types to simplify automatic GUI generation in the workflow layer, but an extension to arbitrary types is readily possible.

6.1 Prototype I: Workflow Actions on the Blockchain

In our first prototype the blockchain stores *workflow actions*. Figures 4 shows a UML class diagram of core classes. We focus on the actions of defining a new workflow model, starting a case and firing a transition. Extensions, for example to cancel a case or unload (mark as deprecated) a workflow model, are readily possible.

As seen in Fig. 4, the prototype defines three kinds of transactions. Every transaction, identified by a universally unique identifier (UUID), contains its creation timestamp and a *PeerCertificate* identifying its originator. The peer certificate is issued and signed by the certificate authority and contains the peer's public key. The originator's private key is used to sign a transaction. A *ModelUpdateTransaction* defines a new workflow model through the associated *PetriNet* object that it carries as payload. The Petri net in turn contains a set of *Place* and *Transition* objects. For simplicity, we forgo versioning of workflow models and updates to running cases, as this is not relevant to the blockchain infrastructure. An *InitCaseTransaction* indicates the launch of a new case for a given workflow model. Cases are identified by a UUID and the name of the Petri net; these attributes are the payload of an *InitCaseTransaction*.

A *FireTransitionTransaction* signals that a work item for a given case, corresponding to a transition in the workflow model, has been completed. It carries as payload an *ActivityInstance* object representing the work item (Fig. 4). The work item contains the case ID, the Petri net name, input data, as well as pre- and post-execution values for output data. It is associated with a *Transition* object of a *PetriNet*. Every transition is in turn associated with an *ActivitySpecification*, which specifies which blockchain node it is destined for (host and port attributes) as well as any data constraints, external method call information, and input and output variables. Work items (*ActivityInstance* objects) provide methods to execute and undo them, to check data constraints,

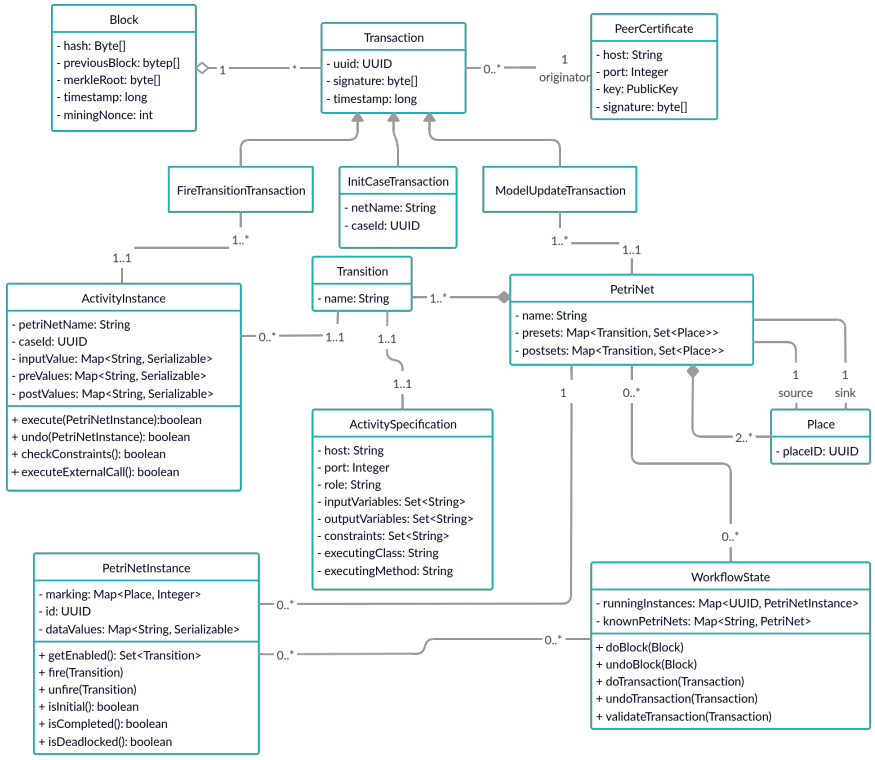

Fig. 4 UML class diagram for prototype I (simplified)

and to execute calls to external methods. Pre-execution values are required for undo ability (Sec. 7).

Upon receipt of an *InitCaseTransaction* or a *FireTransitionTransaction*, the workflow engine initializes or updates the data values and Petri net marking in the workflow instance, described by the *PetriNetInstance* class in Fig. 4. It then identifies newly enabled transitions that are assigned to the local node, creates work items for them and adds these to the local worklist or executes the specified external method calls.

Because the blockchain only stores state changing actions, the workflow engine needs to maintain the workflow state, represented by the *WorkflowState* class in Fig. 4. The workflow state consists of known workflow specifications and the set of running cases, each with their execution states. The execution state for each case is described by a *PetriNetInstance* object, which represents a running case. Each *PetriNetInstance* has attributes for the current data values and the current marking of the associated *PetriNet* object.

Fig. 5 shows the UML sequence diagram for creating and submitting a new transaction. The transaction is created by the worklist user interface when the user completes a work item, which submits it for signing to the *P2PNode* before adding it to the *TransactionService*. The transaction service first vali-

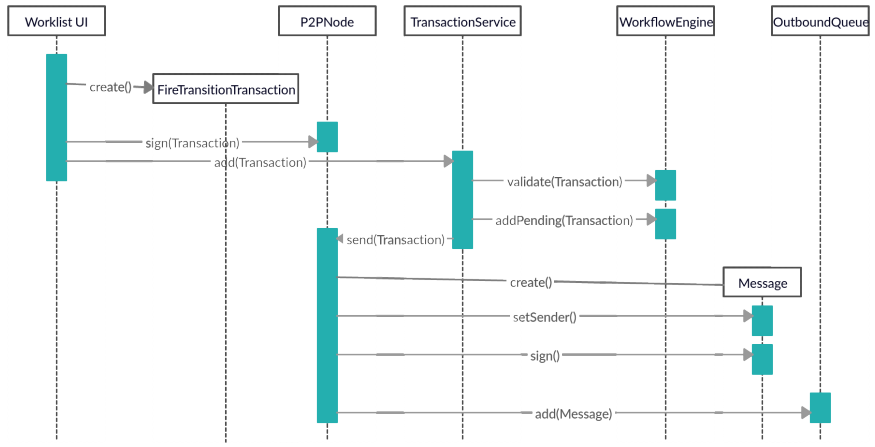

Fig. 5 UML sequence diagram for creating a new transaction (simplified)

dates the transaction with the *WorkflowEngine* and then, if valid, adds it as a pending transaction. It then uses the *P2PNode* to send the new transaction to other nodes. The *P2PNode* creates a new *Message* object, sets the sender information, signs it, and adds the message to the *OutboundQueue* object. When receiving a new transaction from other nodes, the *P2PNode* uses the same "add" method of the *TransactionService*.

Fig. 6 shows the UML sequence diagram for receiving a new block from the P2P network through the inbound message handler; new blocks mined by the local mining service are handled in the same way. The *P2PNode* passes the new block to the local *BlockService*, which first disables the user interface. It then calls its own *append* method. As part of this method, the block service validates the block with the workflow engine; the engine in turn validates each transaction of the block through a call to the workflow state that it maintains (Fig. 4). When the block is validated, it is executed by the workflow engine; each transaction in turn is executed by updating the *ActivityInstance* and the *PetriNetInstance*. Finally, the block service re-enables the user interface.

Fig. 7 shows a screenshot of the prototype, with a list of workflow definitions, running cases, worklisted activities, and pending transactions.

Validity The validation of a *ModelUpdateTransaction* checks that no Petri net with the same name exists in other *ModelUpdateTransactions* in the blockchain or the pending transactions. Validation of a *InitCaseTransaction* checks that a Petri net with the supplied name is defined. Validation of a *FireTransitionTransaction* checks that the Petri net transition of the transaction is enabled, the activity is assigned to the originating node of the transaction, and that no data constraints are violated. For this, the workflow engine executes the pending transactions for that workflow instance to ensure the Petri net transition remains enabled, i.e. the new transaction is not incompatible with any

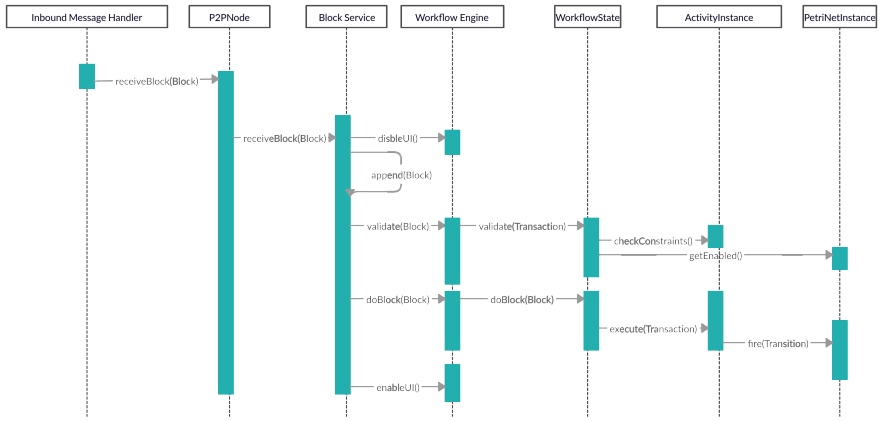

Fig. 6 UML sequence diagram for receiving a new block (simplified)

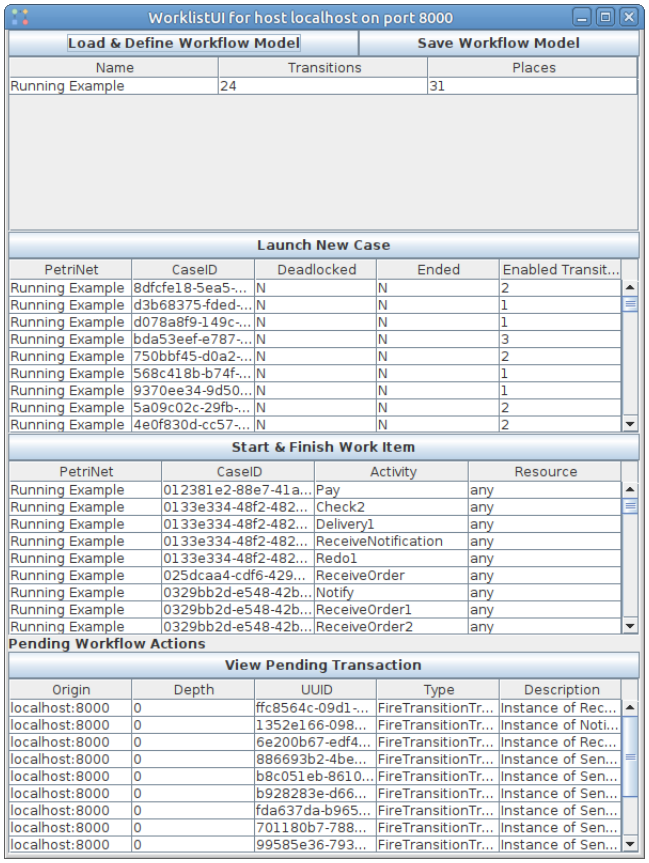

Fig. 7 Screenshot of prototype I

→	Validate(transaction, pendingTransactions)	Pending transactions are those in the transaction pool
→	DoBlock(block)	Announces transactions in the block to the workflow engine for execution
→	UndoBlock(block)	Undoes transactions in the block (during blockchain reorganization)
←	GetPredecessor(block)	Workflow engine gets predecessor block
←	AddTransaction(transaction)	Workflow engine submits new transaction
→	*AddPendingTransaction(transaction)*	*Transaction service notifies engine of pending transaction (optional)*

Table 2 Interface between blockchain infrastructure and workflow engine in prototype I (→ indicates blockchain infrastructure calling workflow engine, ← indicates reverse direction)

of the pending ones for that workflow instance. It then undoes the pending transactions in reverse order to restore the current state.

Interface In general, any blockchain infrastructure performs three functions. It accepts new transactions for inclusion in the blockchain, it provides transactions in new blocks to external applications (as well as invalidated blocks upon a chain reorganization), and it validates transactions. Because transaction validity includes the permissibility of workflow actions, transaction validation requires the workflow engine. These considerations lead to the generic interface in Table 2. First, transaction and block services can call the workflow engine to validate blocks and transactions. Second, the block service passes blocks to the workflow engine for execution (to "do" them). Third, during blockchain reorganization, the block service notifies the engine to invalidate blocks, i.e. to "undo" them. Fourth, in the other direction, the workflow engine can get predecessor blocks from the block service, used for retrieving blocks at specified confirmation depths. Fifth, the workflow engine can add new transactions to the transaction pool. Finally, to gain knowledge of all transactions in the transaction pool, an optional interface for the transaction service to notify the workflow engine of new transactions in the pool is implemented. This is not required for the functioning of the WfMS but is useful from the user perspective (Sec. 7).

6.2 Prototype II: Workflow Instance States on the Blockchain

An alternative to storing workflow activities in transactions on the blockchain is to store complete *workflow instance states*, i.e. data values and Petri net markings. This alternative does not need separate *InitCaseTransaction* and *FireTransitionTransaction*. They are combined into an *InstanceStateTransaction* that represents a complete workflow instance state. The UML class diagram in Fig. 8 shows this change.

This design implies significant changes. First, because activity execution information is not available in the blockchain, data constraints cannot be specified as post-constraints for each activity, but can only be specified for the

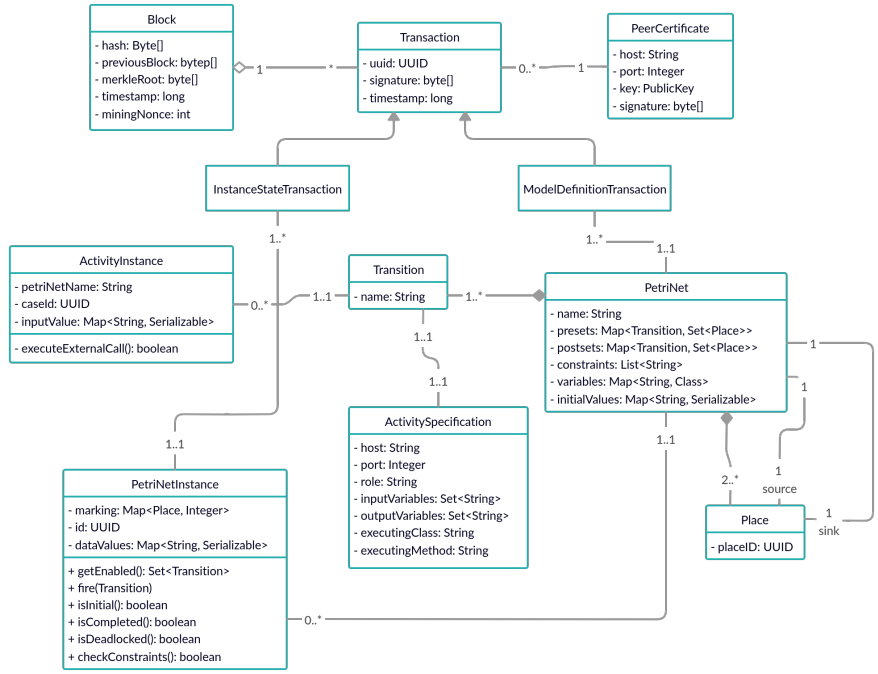

Fig. 8 UML class diagram for prototype II (simplified)

entire state, i.e. they apply to the global case data. Comparing Fig. 4 with 8, the constraint attribute is now with the *PetriNet* class rather than the *ActivitySpecification* class. Second, transactions are not validated by executing and then undoing pending transactions. Consequently, Fig. 8 does not show an "undo" method for the *ActivityInstance* class. Instead, the workflow engine checks that the marking of the workflow instance state in a new transaction is reachable from the marking of the current workflow instance state as well as the markings of the workflow states in all pending transactions. Third, the workflow engine does not need not maintain workflow state information as that is readily available by reading the blockchain backwards from the current chain head. Hence, the class *WorkflowState* is no longer present in the UML class diagram in Fig. 8. This change significantly simplifies the workflow engine. Fourth, the lack of activity execution information means that the user only knows about pending future states, but not which activities bring about those states. Because multiple enabled transitions may lead to the same state, this information cannot be derived from the current and previous state either.

Interface The general interface remains the same, as the blockchain infrastructure performs the same three functions. However, instead of the "Do" and "Undo" ability for transactions in blocks in prototype I, the block service simply notifies the workflow engine when a new block is appended to the head of the chain; the interface is renamed to *UpdateHead*. During blockchain re-

→	ValidateTransaction(transaction, pendingTransactions)	Pending transactions are those in the transaction pool
→	UpdateHead(block)	New blockchain head is added
→	ResetHead(block)	Blockchain head is reset to specified block (during blockchain reorganization)
←	GetPredecessor(block)	Workflow engine gets predecessor block
←	GetDepth(block)	Workflow engine gets confirmation depth of block
←	AddTransaction(transaction)	Workflow engine submits new transaction
→	*AddPendingTransaction(transaction)*	*Transaction service notifies engine of pending transaction (optional)*

Table 3 Interface between blockchain infrastructure and workflow engine (→ indicates blockchain infrastructure calling workflow engine, ← indicates reverse direction)

organization, instead of providing sets of blocks for undo, the block service notifies the workflow engine that the current blockchain head has been reset to a different block and the workflow engine reads the blockchain backwards from the new head to get the new workflow state. Hence, the interface function is renamed to *ResetHead*. The remainder of the interface is unchanged. Table 3 shows the interface for this implementation style.

6.3 Performance

Transaction latency and throughput are not meaningful to compare blockchain deployments as both are affected by block difficulty and block size. While there are limits on these, their possible ranges makes it difficult to compare blockchain performance in a generalizable way. Transaction execution cost or storage cost measured in units of cryptocurrency, as on the Bitcoin or Ethereum blockchains, is also not meaningful for private blockchain deployments as mining is not incentivized and paid for.

Computational Expense A meaningful comparison measure is the (relative) computational expense for validating transactions or blocks and managing the workflow state. We have profiled both prototypes using the VisualVM[2] Java profiling tool. For each prototype we started with an initially empty blockchain. We loaded the running example in Fig. 1 onto the blockchain and mined the model definition transaction to a depth of two blocks, which we assumed as confirmed. We then created a sequence of 1000 workflow transactions as follows. We randomly picked a work item from the work list, simulated its performance, and submitted the corresponding transaction. If the work list was empty, we created a new case. We specified a block difficulty of 19 leading zero bits, a maximum block size of 100 transactions, and a mean delay of 200 milliseconds between submitted transactions. After the first 1000 transactions, we created a second set of 1000 transactions in the same way. We profiled CPU

[2] `https://www.visualvm.org`

performance during this second set of 1000 transactions to ensure a realistic starting point of a non-empty worklist, that all Java classes are loaded in the Java virtual machine, and that any object caches are suitably loaded. We evaluated both prototypes in the same way on the same hardware (quad-core Intel i7 4702HQ laptop) and operating system (Ubuntu 18.04) configuration. Both prototypes required approx. 230 seconds elapsed time to mine the 1000 transactions, for a throughput of approx 4.3 transactions per second. Because raw performance measures are relative to the particular CPU we have normalized the CPU time relative to the mean time of computing a block hash.

Table 4 compares the two prototypes in terms of their relative performance. The table shows the method calls that are identical or similar in purpose for the two prototypes. The table is hierarchically structured. For example, the call to "WorklistUI.addToPendingTx" in line 5 is part of the call to "WorkflowEngine.doBlock" in line 4; the CPU times for line 5 are part of those for line 4. The table is further structured into large sections. Row 1 provides the baseline numbers of calculating a single block hash during mining, rows 2–11 concern the appending of a new block to the blockchain, rows 12–19 reflect the creation of a new transaction, row 20 is the action of digitally signing a transaction, while rows 21 and 22 reflect our test setup where the list of current work items is retrieved and completed work items are removed.

The instance counts in Table 4 shows that in both tests, 34 blocks were created for 1000 transactions. Also in both tests, more than 1000 transactions were submitted, 151 (134) submitted transactions were invalid. This can occur when work items are mutually exclusive, which occurs twice in Fig. 1, at places p_7 and p_{10}. There is no work item removed from the work list for invalid transactions, reflected in the instance count in row 22.

Block mining (row 1) is by far the dominant computational expense and block hashing times are similar for the two prototypes; after all, they use an identical blockchain infrastructure.

Appending a block is about 50% more expensive in prototype II than in prototype I while adding a new transaction is slightly cheaper in prototype II than in prototype I. Further examining these methods shows that user interface operations (rows 5, 6, and 14) consume the bulk of the total CPU time for appending a block or adding a new transaction. User interface methods are similar in their relative CPU time, as the user interface is identical for both prototypes. Comparing the validation times for blocks and transactions (rows 8 and 9) shows that prototype II is computationally more expensive than prototype I. This is also evident when comparing the two prototypes on rows 13 and 14. After accounting for the user interface method in row 14, the remaining CPU times of the totals in row 13 are much higher for prototype II than for prototype I.

In summary, transaction validation is more expensive when only the workflow state is stored on the blockchain (prototype II) than when workflow activities are stored. The reason is that there may be multiple ways in which a workflow state can be transformed into another: Multiple transitions may be enabled and there may be many different sequences of transitions firing to

		Prototype I				Prototype II			
	Method	Total CPU Time (msec)	Instances	CPU Time per Instance (msec)	Relative CPU Time	Total CPU Time (msec)	Instances	CPU Time per Instance (msec)	Relative CPU Time
1	Block.calculateHash	108414	19760259	0.005	1	104621	19835092	0.005	1
2	BlockService.append	3852	34	113.294	20649	6029	34	177.324	33619
3	WorkflowEngine.doBlock	3762	34	110.647	20167	5345	34	157.206	29805
5	WorklistUI.addToPendingTx	1784	1000	1.784	325	2408	1000	2.408	457
6	WorklistUI.removeFromPendingTx	1581	996	1.587	289	2329	1000	2.329	442
8	WorkflowEngine.validateBlock	67.4	34	1.982	361	490	34	14.412	2732
9	WorkflowEngine.validateTransaction	65.9	1000	0.066	12	488	1000	.488	93
10	BlockService.verify	40.8	34	1.200	219	185	34	5.441	1032
11	Block.calculateMerkleRoot	40.2	34	1.182	216	184	34	5.412	1026
12	TransactionService.add	8472	1151	7.361	1342	6171	1134	5.442	1032
13	WorkflowEngine.validateTransaction	2560	1151	2.224	405	1185	1134	1.045	198
14	WorklistUI.addToPendingTx	2071	1000	2.071	377	2408	1000	2.408	457
15	Transaction.verifyOriginator	662	1151	0.575	105	756	1134	0.667	126
16	PeerCertificate.verifySignature	660	1151	0.573	105	555	1134	0.489	93
17	Transaction.verifyContent	574	1151	0.499	91	892	1134	0.784	149
18	CryptoUtils.verifyObjectSignature	570	1151	0.495	90	889	1134	0.784	149
19	BlockService.isTransactionInMainChain	2192	1151	1.904	347	2680	1134	2.363	448
20	P2PNode.signTransaction	432	1151	0.375	68	729	1134	0.643	122
21	WorkflowEngine.getAllWorkItems	68	1151	0.059	11	128	1134	0.113	21
22	WorkflowEngine.removeWorkItem	7.19	849	0.008	2	8.85	866	0.010	2

Table 4 Profiling results for prototype I and II

transform one marking to another. Each of them needs to be checked. When a specific activity, i.e. a transition, is stored on the blockchain, only the enablement of that transition needs to be checked.

Storage Expense Another important performance consideration is the expense of storing transactions on the blockchain. While storage in private blockchain deployments is not paid for through cryptocurrencies, storage is an important aspect of the scalability of the chain.

The UML class diagrams in Figures 4 and 8 show what is serialized into each blockchain transaction or block as payload and metadata. The *ModelUpdateTransaction* is similar for both prototypes and stores a *PetriNet* object with its associated *Transition*, *Place*, and *ActivitySpecification* objects. The size of such a transaction depends on the size of the Petri net and the number of data variables and constraints. Differences between the prototypes occur in the other transaction types. In prototype I, *PetriNetInstance* objects are never stored on the blockchain, only *ActivityInstance* objects are stored in *FireTransitionTransaction* objects, while *PetriNetInstance* objects are maintained and persisted locally by each workflow engine. In prototype II, *ActivityInstance* objects are locally maintained by the workflow engine, while *PetriNetInstance* objects are stored on the blockchain in an *InstanceStateTransaction*. Hence, we expect prototype II to be less efficient for blockchain storage than prototype I.

After running the experimental evaluations described above, the blockchain for prototype I stored 234 *InitCaseTransaction* objects and 1766 *FireTransitionTransaction* objects. The total payload sizes were 62,010 bytes and 1,988,105 bytes, respectively, for a mean size of 265 bytes for the Petri net name and case ID in each *InitCaseTransaction* and of 1126 bytes for the *ActivityInstance* object of each *ModelUpdateTransaction*. The blockchain for prototype II stored 2000 *InstanceStateTransaction* objects that carry a *PetriNetInstance* object as payload. The total payload size was 11,989,445 bytes for on average *PetriNetInstance* object size of 5995 bytes. The large difference in transaction payload size confirms our expectations.

We have not optimized our implementation for speed or storage efficiency. For example, compressed serialization is an obvious way to reduce storage space. Decoupling of the user interface from the workflow engine is an easy way to optimize performance and responsiveness. A careful selection of algorithms and data structures and improved caching can further improve performance. However, the prototype implementations are to be understood as proof-of-feasibility for different architectures and are intended as research vehicles for exploring the implications of blockchain infrastructure (Sec. 7), rather than as fully optimized, production-ready implementations.

6.4 Comparison and Discussion

The two prototypes differ in what is stored in a blockchain transaction: workflow actions or workflow states. Storing workflow actions is the more intuitive

Prototype I Design	**Prototype II Design**
Blockchain stores workflow actions	Blockchain stores workflow states
Workflow engine must provide persistence	Workflow engine need not provide persistence
Case-level and work item-level data constraints	Case-level data constraints only
Informative pending transactions	No informative pending transactions
Low data volume	Higher data volume
Cheap validation	Expensive validation
Suitable for existing workflow engines	Suitable for new workflow engines

Table 5 Major differences between the designs of prototypes I and II

approach but requires a workflow engine that maintains its own state independent of the blockchain. This option is useful for porting existing workflow engines, as they already possess persistence and transaction management capabilities. Additionally, workflow engines are typically built to process workflow actions, rather than entire states. Storing workflow state on the blockchain simplifies development of new workflow engines, as the blockchain infrastructure can be used not only for distribution but also for persistent storage. The ability to simply read complete workflow states off the blockchain eliminates the need for persistent storage by the workflow engine. Both alternatives require the ability to react to changes in the validation state of blocks, which is arguably easier to implement in the second alternative.

Another difference is that prototype II does not make workflow actions, such as work item completion, explicit. Hence, work item level data constraints cannot be enforced or validated on the blockchain. Explicitly recording workflow actions also allows the blockchain infrastructure to provide meaningful information about pending changes to workflow users, whereas full workflow states may not be informative to users. It is of course possible to store both workflow actions and workflow states on the blockchain to achieve the advantages of both designs, but this leads to high storage requirements and significant redundancy.

Finally, storing complete workflow instance states consumes more data on the blockchain than updates only and validation is computationally more expensive, which may be expensive for public chains but is not an issue for private chains.

In summary, unless the rapid and relatively easy development or prototyping of an entirely new WfMS is envisioned, as we have done here, the design of prototype I is preferable for adapting or porting existing WfMS onto a blockchain architecture. Table 5 highlights the differences between the two prototype designs.

7 Blockchain-specific Issues in Workflow Management

Constructing our research prototypes was useful to identify issues presented by proof-of-work blockchain-based WfMS that do not exist in traditional, centralized WfMS, and that have not yet been raised by prior research.

7.1 Latency

Proof-of-work-based blockchains introduce latency. At the very least, this is the time between submitting a transaction to the transaction pool and it being mined. A longer latency is introduced when the preferred confirmation depth requires multiple blocks mined on top of a transaction. For example, the Bitcoin community recommends that transactions are not considered as confirmed and acted upon until six or more blocks are added on top of them; Bitcoin mines a new block approximately every 10 minutes. The Ethereum community recommends to assume confirmation at 10 to 15 blocks, with blocks being mined every 13 seconds. While latency may not be a problem for slow-moving, long-running workflows where progress is measured in days or weeks, it may be a considerable problem for fast-moving, short workflows that must progress within minutes. From the user's perspective, workflow activities in a proof-of-work blockchain-based WfMS can remain pending for a significant amount of time, in contrast to traditional WfMS where actions are completed immediately. Users must be aware of this latency and its impact on the workflow. The user interface of a WfMS must show these transaction states; the user requires insight into the current state for each submitted transaction, as well as for transactions submitted by other users. To provide this information, workflow engines need to be adapted to continuously monitor the blockchain and track the status of all transactions, a significant effort.

7.2 Validating State and Visible State

Workflow transactions go through stages (Table 6), from being accepted to the transaction pool, to being mined into a block, and finally to being considered confirmed and therefore actionable. Only the effects of confirmed transactions should be visible to the user, while the effects of all transactions are used when validating new transactions and new blocks. This distinction leads to two different workflow states, which we call the "visible state" and the "validation state". This distinction is a key difference to traditional WfMS. Understanding the behavior of the WfMS requires the user to have some knowledge about the underlying blockchain infrastructure. In our prototypes we deal with this issue by showing the status of all pending and mined transactions until they are considered confirmed. Only then is the user's worklist, which reflects the "visible state", updated. This can be seen in the bottom part of Fig. 7. Providing information about pending and mined but not yet confirmed transactions (i.e. the validation state) allows the user to understand why subsequent activities may not yet be worklisted, or why certain activities cannot be completed, even though they are worklisted in the visible state. We illustrate possible problems due to this discrepancy by examining the sequence and deferred choice workflow patterns (van der Aalst et al., 2003).

Sequence Consider the sequence of activities t_7 "produce1" and t_8 "check1" in Fig. 1. Both activities are assigned to the same node. While the local workflow

Transaction Stage	Validation	Note
Insert into transaction pool	Validate against transactions in chain and transaction pool	Reject invalid transactions but do not remove their work item from worklist; allow user to retry. Remove work items for valid transactions from worklist and report as pending.
Insert to head of chain	Validate against transactions in chain	Reject blocks with invalid transactions. No changes to the worklist.
Reach confirmation depth of chain		Consider work items for transactions in block as done, and remove them from worklist. Create work items for newly enabled activities.

Table 6 Transaction stages, validation points, and workflow actions

engine and user know that "produce1" has been completed, "check1" cannot be worklisted until "produce1" is considered confirmed. Or can it? One can imagine a speculative execution of a workflow where the local workflow engine worklists activity "check1" immediately, at the risk of having to undo it at a later stage should activity "produce1" not be accepted by the consensus chain or invalidated later. This is a design choice for the particular WfMS. For this, each node's workflow engine must track the status of its own submitted transactions. In case a transaction is removed from the blockchain or transaction pool, e.g. due to conflicts with other transactions during chain reorganization, it must be undone locally. Speculative execution has not been used to address confirmation latency of blockchain-based WfMS and is an interesting direction for future research.

Deferred Choice Consider activity t_8 "check1" in Fig. 1, which is followed by place p_7 and either activity t_9 "deliver1" or t_{10} "redo1". After "check1" is completed and confirmed, both "deliver1" and "redo1" are worklisted to allow the user the choice of which to perform, based on the results of "check1". When "deliver1" completes, "redo1" should be withdrawn, and vice versa. When both activities are assigned to the same node, as in our case, it may make sense to withdraw "redo1" as soon as execution of "deliver1" is completed and submitted to the transaction pool, as this corresponds to the local user's understanding. However, this may conflict with the explicit confirmation depth requirement of that user, which may be one or more blocks. From that perspective, "redo1" should not yet be withdrawn. Our prototypes implement the latter approach and do not withdraw "redo1" from the worklist; completion of "deliver1" is reported as pending to the user. Of course, completion of "redo1" cannot be added as a new transaction. Hence, despite "redo1" being worklisted, the user is presented with an error notice upon its completion (not upon its start, as validation is done only when the engine attempts to add a *FireTransitionTransaction* to the transaction pool). When "deliver1" and "redo1" are allocated to different nodes, the users may not be aware of the deferred choice situation or the execution status of the other activity. In this situation, "redo1" should not be withdrawn when "deliver1" is submitted to

the transaction pool: It may be that "deliver1" does not get mined into the chain, perhaps because the local node mines on what will turn out to be a side branch, or "redo1" may be mined into the chain because it arrives at the winning miner before "deliver1" does. Without understanding the underlying blockchain infrastructure, this behavior will be confusing to a user as it differs greatly from that of a traditional WfMS.

Confusion can also arise because what is considered confirmed and therefore visible and actionable depends on each node's required confirmation depth of transactions. A user on one node may consider a particular state to permit execution of a following activity, while a user on another node considers that state as insufficiently confirmed. In sequential workflows, this might lead to tensions as one user believes another is delaying the workflow unnecessarily. In a deferred choice situation, this might lead to competitive behavior as one user can always make the choice before the other.

In summary, it is easy to see how the consensus mechanism in proof-of-work blockchains can lead to confusion if users do not have a good understanding of the principles of blockchain infrastructure. To address this, blockchain-based WfMS will require considerable adaptation to user interfaces, as we have begun to show in our prototypes, as well as user training.

7.3 Confirmed is not Committed: Undo Required

Sec. 4 described how a transaction, even with mined blocks on top of it, may still be invalidated because of chain reorganization; there is no final commit in a proof-of-work blockchain. Because of this, blockchain-based WfMS require the ability to "undo" transactions. In prototype I we implement "undo" by storing the before-values of all outputs of a work item. An interface of the workflow engine allows the block service to ask for "undo" of entire blocks. In prototype II, the undo is performed by resetting the blockchain head and reading back the workflow state from the new head.

User issues Consider a deferred choice of two activities A and B that are assigned to different nodes. Due to latency, it is possible that activity A is completed and mined into a block, while activity B is also completed and mined into a block. Both blocks are the head of the main branch on their originating nodes, i.e. from the local user's perspectives the work items are completed and the corresponding transactions may even be considered confirmed. However, one of them, assume activity B, will be eventually be "undone". Does the user need to be notified of the undo of the transaction so that she can take appropriate action? If so, when and how should the workflow engine notify the user and what information should it provide? Our prototypes add the undone transaction into the list of pending transactions shown to the user, but without highlighting this or raising an alarm for the user. Other implementations could take a more active approach. To take this example further, consider an activity X to be performed prior to A or B on some third node. It may be

possible (although unlikely) that X, and therefore *both* A and B are "undone" so that both workflow users are presented again with deferred choice of A or B. Both users were convinced that the workflow state includes completion of A or B, respectively, yet both activities are worklisted again. In summary, the eventual consistency approach in proof-of-work blockchains requires users to be made aware of the state of the workflow and each transaction, and be able to understand and make sense of non-intuitive changes to the workflow states.

External effects In contrast to purely financial transactions such as cryptocurrency transfers (or other virtual transactions, e.g. transfer of ownership), activities in business processes represent considerable human work and other resource consumption. In the best case, undoing workflow transactions wastes this work and the consumed resources. In the worst case, they may not be undoable at all. While financial transactions are reversible by crediting and debiting appropriate accounts, the performance of workflow activities may represent substantial and permanent changes of state in the physical world. Applying greater confirmation depth to transactions merely reduces the problem, but does not eliminate it.

Compensation To address external effects that cannot be undone, the idea of compensating activities or workflow fragments may be useful. Compensation in workflows is a complex issue (e.g. Eder and Liebhart, 1996; Grefen et al., 2001; Acu and Reisig, 2006) but in the blockchain context it raises further questions such as when to worklist compensation activities, whether compensation activities must pre-empt other work items for that case, whether compensation should be done on the validating state, the visible state, or both. For example, consider the blockchain in Fig. 2 and a user with a confirmation depth requirement of one block. Assume again that block 4 is a new block to be added and block 3b is the head of the main branch. During chain reorganization, transaction 31b in block 3b is invalidated. However, the user has not yet "seen" transaction 31b, as her visible state is only up to block 2b. To offer a compensating activity for 31b will be confusing. A compensating activity for the invalidated transaction 21b may be appropriate, but which state, the validating or the visible or both, should be updated with the outputs of this activity? Moreover, the invalidated transactions may refer to activities performed on other nodes, but chain reorganization is a local issue. Should compensating activities be inserted into the original nodes's worklist; yet, that node may not experience the chain reorganization? Should results of compensating activities be confined to the local state only (what about state consensus?) or should they be broadcast on the blockchain (even though some nodes do not undergo a chain reorganization)?

Blockchain WfMS based on smart contracts avoid the "undo" requirement as the blockchain will, upon chain reorganization, automatically restore a previous smart contract state because the contract state is encoded on the blockchain itself. However, this does nothing to address the user issues, external

effects, and compensation problems highlighted here, which remain a problem also for that architecture.

In summary, the "undo" required by proof-of-work-based WfMS highlights ambiguities in execution order; it complicates user's understanding of the workflow state and requires user understanding of the underlying blockchain architecture. "Undo" may not be possible for some activities or may lead to considerable wasted resources and effort for others. Addressing the problem with compensation may be appealing but raises further difficult questions.

7.4 Data Dependencies

Transactions in the transaction pool and transactions within the same block are considered unordered because timestamps in a distributed blockchain are unreliable as there is no central clock. Consider two parallel activities, for example t_{17} "Send Order 1" and t_{18} "Send Order 2", that both write to the same data variable. The same user performs "Send Order 1" followed by "Send Order 2". She therefore expects the data variable to have a certain value. In the absence of control-flow dependencies, the transactions for both work items are mined into the same block. As the block and its transactions arrive at nodes, including the originating node, the workflow engines must execute "Send Order 1" and "Send Order 2", but in what order? The issue is similarly present when undoing workflow actions. During chain reorganization, the system cannot decide which activity to undo first. Blockchains resolve the ambiguity by executing transactions in block order or by otherwise fixing the order. In any case, the order in which they are executed may not match the order in which they were performed and the value of the data variable may therefore not match the user's expectations. One might argue that even in traditional WfMS the execution order is, in the absence of control-flow dependencies, arbitrary and that, in the presence of data-dependencies, the workflow designer ought to have specified control-flow dependencies. However, in a traditional WfMS the execution order is determined by the user's performance and thus matches the user's expectations.

8 Discussion and Conclusions

Previous work on blockchain-based WfMS has focused on smart contracts for the Ethereum blockchain that implement workflow engines on-chain. In contrast, we have worked with traditional off-chain workflow engines in our research prototypes. The workflow engines include case data management, data constraints, and local resource management using standard workflow engine designs. We have highlighted issues around workflow state visibility, latency, transaction confirmation, and data dependencies that have not been considered in prior research. Our research prototypes are not meant for production use; they are research tools to explore blockchain-based workflow management in a controlled environment. However, the problems we have identified

and our recommendations below are generalizable beyond our implementations to proof-of-work blockchain-based WfMS in general. They also apply to smart contract-based implementations, as they stem from the properties of proof-of-work chains, not from the specific blockchain architecture or our implementations.

8.1 Recommendations

We have shown that proof-of-work blockchain infrastructure introduces unique issues that require both user interface adaptations as well as user awareness and training. We make the following recommendations.

User interfaces The effects of proof-of-work blockchains cannot be hidden from the user. Rather than trying to hide the infrastructure from the user, we recommend that WfMS designers provide full visibility. This includes aspects such as tracking the status of locally and remotely submitted transactions and indicating their confirmation depth, as we have done in our prototypes. User interfaces should also provide informative and constructive feedback and alerts, for example, when users attempt to perform an activity that is incompatible with pending activities, or when a chain reorganization takes place and leads to activities that were assumed to be completed to be worklisted again.

User education The recommendations for user interfaces are only useful if users are aware of at least the general principles of proof-of-work consensus. Again, hiding the effects of this infrastructure is not possible, so that WfMS users must be educated on transaction states, causes of latency in the system, and the possibility and principles of chain reorganizations. Users must not only be trained on the WfMS, but also on the concepts of blockchains and the proof-of-work consensus mechanism, both of which are complex concepts. Hence, this may pose considerable challenges in practice and detract from users' actual work.

Process designs One way to mitigate the effects of blockchain infrastructure is to reduce the number of case transfers between nodes. We recommend that workflow designers consider the sub-contracting pattern (van der Aalst, 1999), which decomposes activities to sub-workflows of which all activities are assigned to the same node. A hybrid architecture of local WfMS that are joined by a blockchain infrastructure can ensure that only the high-level inter-organizational workflow is affected by the effects of the blockchain infrastructure. Lessons learned from distributed WfMS (Sec. 3.1) and the use of projection inheritance (van der Aalst, 2002, 2003) to ensure behavioral correctness apply to such hybrid architectures. Workflow designers should also consider compensation activities to be performed when a transaction cannot be undone (e.g. Eder and Liebhart, 1996; Grefen et al., 2001; Acu and Reisig, 2006).

Blockchain Use A key motivator for using blockchain-based WfMS is the lack of trust among process participants. While proof-of-work blockchains, in particular Ethereum, are popular with WfMS researchers, other blockchain technologies exist that make the same validity and consensus guarantees but do not exhibit the drawbacks of proof-of-work blockchains. For example, PBFT-based systems (Practical Byzantine Fault Tolerance) systems (Androulaki et al., 2017; Sousa et al., 2018) do not scale well with the number of nodes but offer very low latency and finality of consensus. Hence, they may be useful for applications with a small number of organizations and and workflows that require low latency and final consensus (Vukolić, 2015).

8.2 Smart Contracts versus Application Code

Prior work on blockchain-based WfMS (Sec. 3.3) uses smart contracts. Smart contracts provide code integrity and visibility/transparency, as the code is part of the blockchain. Additionally, there is no need to call outside the blockchain layer for transaction validation, as we do in our architecture. The disadvantages are the need to re-develop existing application logic. The strong focus on control flow neglects implementation aspects typically handled by workflow engines such as data management, data transformations, constraints, external services, scripting, decision tables, organizational data management, role resolution, user interfaces, and others. By not porting and reusing traditional workflow engines, implementing these aspects leads to considerable effort for a smart contract architecture, which may be exacerbated due to limitations of the smart contract language instruction set.

In our architecture, the blockchain is treated simply as a trusted infrastructure layer. It serves only to record and share the state of a workflow execution and achieve consensus on the validity of that state. This architecture offers not only the ability to adapt existing workflow engines using simple interfaces (Sec. 6), avoiding re-implementation effort and relying on proven technologies, but also offers more freedom to implement features that may not be possible in the blockchain execution environment. Implementing application logic off-chain means that developers have access to familiar programming languages, code libraries and development tools. One drawback is that transaction validation must call back to the application logic. Unlike smart contracts, performing validation in off-chain logic places the onus on developers to ensure identical results if nodes use different workflow engines. However, off-chain validation allows developers to develop against a behavioural specification, e.g. workflow net semantics, without specifying the exact algorithms or implementation to be used: The architecture can use a heterogeneous set of workflow engines, each best suited to a particular node's requirements. Table 8.2 lists advantages and disadvantages of the two architectures.

Workflow Engine on Blockchain	**Workflow Engine off Blockchain**
Requires re-development of workflow engine	Can adapt existing workflow engines
Separates workflow engine from external services	Workflow engine remains integrated with external services
Separation of workflow logic from transaction validation	Workflow logic is part of transaction validity; requires call-back to workflow engine for transaction validation
Code integrity & visibility	Develop against a behavioural specification
Ensures identical behavior for all peers	Behavior must be independently validated by each peer
No design freedom for peers	Allows heterogeneous engine implementations
May be limited by blockchain execution environment	No implementation limitations
May be limited in integrating off-chain components	Few limitations to integrate off-chain components

Table 7 Architectural options for blockchain-based workflow management systems

8.3 Limitations and Future Work

While we have identified many issues around proof-of-work blockchain-based WfMS and have made recommendations to address them, our work also shows limitations, which we view as avenues for future research. We see both technical as well as empirical research opportunities.

Technical research opportunities On the technical side, our work has shown potential for further refinement and exploration of architectural options, in particular the following topics:

- Designing processes to minimize the effects of blockchain infrastructure.
- Investigating speculative execution of local activities with possible "undo". Can speculative execution protocols from other areas in computer science be used and adapted for workflow management?
- Using compensation activities or compensation workflow fragments to support improved "undo" of transactions.
- Porting existing workflow engines, such as the open-source YAWL system (ter Hofstede et al., 2009), to blockchain infrastructure.
- Extending the architecture from a $1:1$ relationship between blockchain nodes and workflow engines to an $n:m$ relationship.
- Implementing BFT-based blockchains for WfMS to identify architectural design issues and implications for WfMS and their users.

Empirical research opportunities Neither this nor prior work has advanced beyond prototypes and feasibility studies into production settings. Hence, little large scale or in-depth empirical research on user and organizational issues is available. Our recommendations, as they affect WfMS users, will eventually require empirical support. One of the central points is our recommendation

to identify effective ways of communicating workflow and transaction state to users, and to make users understand the specific implications of using proof-of-work blockchain infrastructure. Supporting this requires observational or experimental work with users and explorations of different WfMS user interface designs, both in a field as well as a laboratory setting. Similarly, the question of whether our architecture can be applied effectively in production settings requires empirical work. Investigating the applicability and feasibility of our architecture needs in-depth case studies into blockchain deployments in inter-organizational settings.

We believe that application-specific blockchains, like the architecture investigated here, are easier to deploy for stand-alone, ad-hoc applications where the participating organizations have not yet invested into a common, generic blockchain platform. Such situations occur when organizations need to complete workflows in a project setting, instead of a permanent, ongoing basis. One example of such a situation is the permitting of exploration or extraction in the natural resources industry, such as mining and petroleum. Such a process frequently involves multiple stakeholders with different interests and a reasonably structured process of consultation, negotiation, etc.

Finally, Mendling et al. (2018) point out the "people" factor in adopting blockchain-based WfMS, which they view as an acceptance problem from the enterprise perspective. Through our research, we have identified specific user-focused challenges, such as interface design and user education in blockchain technology, solutions to which will help gain user acceptance.

To conclude, this paper has described an architecture and two implementations for a blockchain-based WfMS that has not yet seen research attention. Our research shows that workflow engines do not need to be implemented using smart contracts but that traditional workflow engines and the modelling languages they support, can be readily adapted to fit onto a blockchain infrastructure. The interfaces between workflow engines and blockchain infrastructure are simple, and independent of the semantics of the workflow description language. Our work has also highlighted many aspects in which blockchain-based WfMS differ from traditional systems. We have discussed implications of these differences and demonstrated how we they can be addressed in our prototype work.

Conflicts of Interest

On behalf of all authors, the corresponding author states that there is no conflict of interest.

References

Acu, B. and Reisig, W. (2006). Compensation in workflow nets. In *Intnl. Conference on Application and Theory of Petri Nets*, pages 65–83. Springer.

Alonso, G., Mohan, C., Günthör, R., Agrawal, D., El Abbadi, A., and Kamath, M. (1995). Exotica/FMQM: A persistent message-based architecture for distributed workflow management. In *Information Systems Development for Decentralized Organizations*, pages 1–18. Springer.

Androulaki, E., Cachin, C., De Caro, A., Sorniotti, A., and Vukolic, M. (2017). Permissioned blockchains and Hyperledger Fabric. *ERCIM News*, 110:9–10.

Atluri, V., Chun, S. A., Mukkamala, R., and Mazzoleni, P. (2007). A decentralized execution model for inter-organizational workflows. *Distributed and Parallel Databases*, 22(1):55–83.

Bauer, T. and Dadam, P. (1997). A distributed execution environment for large-scale workflow management systems with subnets and server migration. In *Proceedings of the Second IFCIS International Conference on Cooperative Information Systems*, pages 99–108. IEEE Computer Society.

Bauer, T. and Dadam, P. (2000). Efficient distributed workflow management based on variable server assignments. In Wangler, B. and Bergman, L., editors, *Advanced Information Systems Engineering, 12th International Conference, Proceedings*, volume 1789 of *LNCS*, pages 94–109. Springer.

Castro, M. and Liskov, B. (2002). Practical byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, 20(4):398–461.

Chebbi, I., Dustdar, S., and Tata, S. (2006). The view-based approach to dynamic inter-organizational workflow cooperation. *Data Knowl. Eng.*, 56(2):139–173.

Ciccio, C. D., Cecconi, A., Dumas, M., Garcia-Banuelos, L., Lopez-Pintado, O., Lu, Q., Mendling, J., Ponomarev, A., Tran, A. B., and Weber, I. (2019). Blockchain support for collaborative business processes. *Informatik Spektrum*, 42(3):182–190.

Das, S., Kochut, K., Miller, J., Sheth, A., and Worah, D. (1997). ORBWork: A reliable distributed CORBA-based workflow enactment system for METEOR2. In *Proc. of the 23nd. Intnl. Conference on Very Large Data Bases, Athens, Greece*.

Dogac, A., Gokkoca, E., Arpinar, S., Koksal, P., Cingil, I., Arpinar, B., Tatbul, N., Karagoz, P., Halici, U., and Altinel, M. (1998). Design and implementation of a distributed workflow management system: Metuflow. In *Workflow Management Systems and Interoperability*, pages 61–91. Springer.

Eder, J. and Liebhart, W. (1996). Workflow recovery. In *Proceedings First IFCIS International Conference on Cooperative Information Systems*, pages 124–134. IEEE.

Eder, J. and Panagos, E. (1999). Towards distributed workflow process management. In Bussler, C., Grefen, P. W. P. J., Ludwig, H., and Shan, M., editors, *Proceedings of the Workshop on Cross-Organisational Workflow Management and Co-ordination*, volume 17 of *CEUR Workshop Proceedings*.

Fakas, G. J. and Karakostas, B. (2004). A peer to peer (P2P) architecture for dynamic workflow management. *Information & Software Technology*, 46(6):423–431.

Falazi, G., Hahn, M., Breitenbücher, U., and Leymann, F. (2019a). Modeling and execution of blockchain-aware business processes. *SICS Software-*

Intensive Cyber-Physical Systems, 34(2-3):105–116.

Falazi, G., Hahn, M., Breitenbücher, U., Leymann, F., and Yussupov, V. (2019b). Process-based composition of permissioned and permissionless blockchain smart contracts. In *2019 IEEE 23rd International Enterprise Distributed Object Computing Conference (EDOC)*, pages 77–87. IEEE.

Fridgen, G., Radszuwill, S., Urbach, N., and Utz, L. (2018). Cross-organizational workflow management using blockchain technology - towards applicability, auditability, and automation. In *51st Hawaii International Conference on System Sciences HICSS*. AIS Electronic Library.

García-Bañuelos, L., Ponomarev, A., Dumas, M., and Weber, I. (2017). Optimized execution of business processes on blockchain. In Carmona, J., Engels, G., and Kumar, A., editors, *Business Process Management - 15th International Conference, BPM, Proceedings*, volume 10445 of *Lecture Notes in Computer Science*, pages 130–146. Springer.

Geppert, A. and Tombros, D. (1998). Event-based distributed workflow execution with EVE. In *Middleware98*, pages 427–442. Springer.

Gillmann, M., Weißenfels, J., Weikum, G., and Kraiss, A. (2000). Performance and availability assessment for the configuration of distributed workflow management systems. In Zaniolo, C., Lockemann, P. C., Scholl, M. H., and Grust, T., editors, *EDBT 2000, 7th International Conference on Extending Database Technology, Proceedings*, volume 1777 of *Lecture Notes in Computer Science*, pages 183–201. Springer.

Grefen, P., Aberer, K., Hoffner, Y., and Ludwig, H. (2000). CrossFlow: Cross-organizational workflow management in dynamic virtual enterprises. *Computer Systems Science and Engineering*, 15(5):277–290.

Grefen, P., Vonk, J., and Apers, P. (2001). Global transaction support for workflow management systems: from formal specification to practical implementation. *The VLDB Journal*, 10(4):316–333.

Härer, F. (2018). Decentralized business process modeling and instance tracking secured by a blockchain. In Bednar, P. M., Frank, U., and Kautz, K., editors, *26th European Conference on Information Systems ECIS*, page 55. AIS Electronic Library.

Hukkinen, T., Mattila, J., Seppälä, T., et al. (2017). Distributed workflow management with smart contracts. Technical report, The Research Institute of the Finnish Economy.

Jin, L., Casati, F., Sayal, M., and Shan, M. (2001). Load balancing in distributed workflow management system. In Lamont, G. B., editor, *Proceedings of the 2001 ACM Symposium on Applied Computing (SAC)*, pages 522–530. ACM.

Ladleif, J., Weske, M., and Weber, I. (2019). Modeling and enforcing blockchain-based choreographies. In *International Conference on Business Process Management*, pages 69–85. Springer.

López-Pintado, O., García-Bañuelos, L., Dumas, M., and Weber, I. (2017). Caterpillar: A blockchain-based business process management system. In Clarisó, R., Leopold, H., Mendling, J., van der Aalst, W. M. P., Kumar, A., Pentland, B. T., and Weske, M., editors, *Proceedings of the BPM Demo*

Track co-located with 15th International Conference on Business Process Modeling, volume 1920 of *CEUR Workshop Proceedings*.

Mendling, J., Weber, I., van der Aalst, W. M. P., vom Brocke, J., Cabanillas, C., et al. (2018). Blockchains for business process management - challenges and opportunities. *ACM Trans. Management Inf. Syst.*, 9(1):4:1–4:16.

Miller, J. A., Palaniswami, D., Sheth, A. P., Kochut, K., and Singh, H. (1998). Webwork: Meteor$_2$'s web-based workflow management system. *J. Intell. Inf. Syst.*, 10(2):185–215.

Miller, J. A., Sheth, A. P., Kochut, K. J., and Wang, X. (1996). CORBA-based run-time architectures for workflow management systems. *Journal of Database Management (JDM)*, 7(1):16–27.

Muth, P., Wodtke, D., Weißenfels, J., Dittrich, A. K., and Weikum, G. (1998). From centralized workflow specification to distributed workflow execution. *J. Intell. Inf. Syst.*, 10(2):159–184.

Prybila, C., Schulte, S., Hochreiner, C., and Weber, I. (2020). Runtime verification for business processes utilizing the bitcoin blockchain. *Future Generation Computer Systems*, 107:816–831.

Reichert, M. and Bauer, T. (2007). Supporting ad-hoc changes in distributed workflow management systems. In Meersman, R. and Tari, Z., editors, *OTM Confederated International Conferences CoopIS, DOA, ODBASE, GADA, and IS, Proceedings, Part I*, volume 4803 of *Lecture Notes in Computer Science*, pages 150–168. Springer.

Reichert, M., Rinderle, S., and Dadam, P. (2003). Adept workflow management system. In *International Conference on Business Process Management*, pages 370–379. Springer.

Rimba, P., Tran, A. B., Weber, I., Staples, M., Ponomarev, A., and Xu, X. (2017). Comparing blockchain and cloud services for business process execution. In *2017 IEEE International Conference on Software Architecture, ICSA*, pages 257–260. IEEE Computer Society.

Sousa, J., Bessani, A., and Vukolic, M. (2018). A byzantine fault-tolerant ordering service for the Hyperledger Fabric blockchain platform. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 51–58. IEEE.

ter Hofstede, A. H., van der Aalst, W. M., Adams, M., and Russell, N. (2009). *Modern Business Process Automation: YAWL and its support environment.* Springer Science & Business Media.

van der Aalst, W. (2000). Loosely coupled interorganizational workflows: Modeling and analyzing workflows crossing organizational boundaries. *Information & Management*, 37(2):67–75.

van der Aalst, W. M., ter Hofstede, A. H., Kiepuszewski, B., and Barros, A. P. (2003). Workflow patterns. *Distributed and parallel databases*, 14(1):5–51.

van der Aalst, W. M. P. (1998). The application of petri nets to workflow management. *Journal of Circuits, Systems, and Computers*, 8(1):21–66.

van der Aalst, W. M. P. (1999). Process-oriented architectures for electronic commerce and interorganizational workflow. *Inf. Syst.*, 24(8):639–671.

van der Aalst, W. M. P. (2002). Inheritance of interorganizational workflows to enable business-to-business. *Electronic Commerce Research*, 2(3):195–231.

van der Aalst, W. M. P. (2003). Inheritance of interorganizational workflows: How to agree to disagree without loosing control? *Information Technology and Management*, 4(4):345–389.

van der Aalst, W. M. P. and Weske, M. (2001). The P2P approach to interorganizational workflows. In Dittrich, K. R., Geppert, A., and Norrie, M. C., editors, *Advanced Information Systems Engineering, 13th International Conference, CAiSE, Proceedings*, volume 2068 of *Lecture Notes in Computer Science*, pages 140–156. Springer.

Vossen, G. and Weske, M. (1999). The WASA2 object-oriented workflow management system. In Delis, A., Faloutsos, C., and Ghandeharizadeh, S., editors, *SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data*, pages 587–589. ACM Press.

Vukolić, M. (2015). The quest for scalable blockchain fabric: Proof-of-work vs. BFT replication. In *International workshop on open problems in network security*, pages 112–125. Springer.

Weber, I., Xu, X., Riveret, R., Governatori, G., Ponomarev, A., and Mendling, J. (2016). Untrusted business process monitoring and execution using blockchain. In Rosa, M. L., Loos, P., and Pastor, O., editors, *Business Process Management - 14th International Conference, BPM, Proceedings*, volume 9850 of *Lecture Notes in Computer Science*, pages 329–347. Springer.

Weigand, H. and van den Heuvel, W. (2002). Cross-organizational workflow integration using contracts. *Decision Support Systems*, 33(3):247–265.

Yan, J., Yang, Y., and Raikundalia, G. K. (2006). Swindew-a p2p-based decentralized workflow management system. *IEEE Trans. Systems, Man, and Cybernetics, Part A*, 36(5):922–935.