Workflow Management on BFT Blockchains

Joerg Evermann[1] and Henry Kim[2]

[1] Memorial University of Newfoundland, St. John's, Canada
jevermann@mun.ca
[2] York University, Toronto, Canada
hkim@york.ca

Abstract. Blockchain technology has been proposed as a new infrastructure technology for a wide variety of novel applications. Blockchains provide an immutable record of transactions, making them useful when business actors do not trust each other. Their distributed nature makes them suitable for inter-organizational applications. However, proof-of-work based blockchains are computationally inefficient and do not provide final consensus, although they scale well to large networks. In contrast, blockchains built around Byzantine Fault Tolerance (BFT) algorithms are more efficient and provide immediate and final consensus, but do not scale well to large networks. We argue that this makes them well-suited for workflow management applications that typically include no more than a few dozen participants but require final consensus. In this paper, we discuss architectural options and present a prototype implementation of a BFT-blockchain-based workflow management system (WfMS).

Keywords: Byzantine fault tolerance · blockchain · workflow management · interorganizational workflow · distributed workflow

1 Introduction

Inter-enterprise business processes may include stakeholders in adversarial relationships, that nonetheless have to jointly complete process instances. Trust in the current state of a process instance and correct execution of activities by other stakeholders may be lacking. Blockchain technology can help in such situations by providing a trusted, distributed workflow execution infrastructure.

A blockchain cryptographically signs a series of blocks, containing transactions, so that it is difficult or impossible to alter earlier blocks in the chain. In a distributed blockchain, actors independently validate transactions, add them to the blockchain, and replicate the chain across different nodes. The independent and distributed nature of actors requires finding a consensus regarding the validity and order of transactions and blocks. In workflow execution, it is important that actors agree on the "state of work" as this determines the set of next valid activities in the process. Hence, it is natural to use blockchain transactions to describe workflow activities or workflow states.

Blockchain technology admits many different system designs, and WfMS can be implemented in many different ways on blockchain infrastructure. In this paper, we explore these architectural options. Specifically, we focus on the interface between the blockchain and the workflow engine. In contrast to prior work, which has focused on transaction ordering through proof-of-work consensus, we examine the use of consensus protocols based on algorithms for Byzantine Fault Tolerance (BFT).

Contribution We describe a prototype WfMS system as a proof-of-concept implementation for an architecture that has not yet seen attention in the literature. First, in contrast to earlier work (Sec. 2) we do not use smart contracts to implement model-specific workflow engines. We show that generic or existing workflow engines can be readily adapted to fit onto a blockchain infrastructure and that smart contracts are not required. Second, as recommended, but not implemented by [16], we show how a BFT-based blockchain can be used as workflow management infrastructure. We describe the implementation of a blockchain-based WfMS that has served as our tool to investigate design choices, problems and solutions in this research area. While our prototype is an important demonstration of feasibility, our main contribution is in the identification and discussion of the different architectural choices.

The remainder of the paper is structured as follows. Section 2 reviews related work on blockchain-based WfMS. We then describe the principles of distributed blockchains with a focus on BFT-based consensus (Sec. 3). Section 5 presents our prototype implementation. The final Sec. 6 discusses implications of BFT-based blockchain technology for WfMS and an outlook to future work.

2 Related Work

Blockchain-based workflow management has only recently received research attention [13]. The main research challenges are around integration of blockchain infrastructure into WfMS and ensuring correctness and security of the workflow execution [13]. A number of prototype implementations have been presented, focusing on the use of "smart contracts". A smart contract is a software application that is recorded on the blockchain. This application "listens" for relevant transactions sent to it and executes application logic upon receipt of a transaction. For example, the Ethereum blockchain has Turing-complete virtual machine (VM) for smart contracts and compilers for different programming languages.

Driven by a financial institution, a prototype workflow implementation using smart contracts on the Ethereum blockchain offers digital document flow in the import/export domain [5, 6]. The project demonstrates significantly lowered process cost, increased transparency, and trust among trading partners.

A blockchain-based workflow research project in the real-estate domain [11], also using the Ethereum blockchain and smart contracts, notes that the decentralized nature of blockchains and the lack of a central agency will make it difficult for regulators to enforce obligations and responsibilities of trading partners.

A complete WfMS, including collaborative workflow modelling and model instantiation, uses models as contracts between collaborators [8]. 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. The authors note that the usefulness of the approach is limited by block size limits on the blockchain and the latency between new blocks [8].

Another implementation of blockchain-based workflow execution [17, 18] uses smart contracts on the Ethereum blockchain either as a choreography monitor, where the smart contract monitors execution status and validity of workflow messages against a process model, or as an active mediator, where the smart contract "drives" the process by sending and receiving messages according to a process model. BPMN models are translated into smart contracts. Local Ethereum nodes monitor the blockchain for relevant messages from the smart contract and create messages for the smart contract. Transaction cost and latency are recognized as important considerations in the evaluation of the approach. A comparison between the public Ethereum blockchain and the Amazon Simple Workflow Service cloud-based environment shows blockchain-based costs to be two orders of magnitude higher than a traditional infrastructure [14]. Hence, optimizing the space and computational requirements for smart contracts is important [7]. BPMN models are first translated to Petri Nets, for which minimizing algorithms are known. The minimized Petri nets are then compiled into smart contracts, achieving up to 25% reduction in transaction cost [17, 18], while also significantly improving the throughput time. Building on lessons learned from [17, 18], Caterpillar is an open-source blockchain-based business process management system [12]. Developed in Node.js it uses standard Ethereum tools, like the Solidity compiler solc and the Ethereum client geth, to provide a distributed execution environment for BPMN-based process models.

After examining different blockchain consensus mechanisms in terms of termination time and fault tolerance, BFT-based consensus is recommended for business process executions [16]. The authors propose an architecture that decouples the workflow system from the blockchain by having each blockchain node listen for workflow-relevant events to be passed to the workflow layer. However, the authors do not present an implementation of their proposal.

3 Blockchains

A blockchain consists of blocks of transactions. A transaction can be any kind of content. Information integrity is maintained by applying a hash function to the content of each block, which also contains the hash of the previous block in the chain. Hence, altering a block requires changing all following blocks. In a typical distributed blockchain, nodes are connected using a peer-to-peer network topology. New transactions may originate on any peer and must be placed into new blocks. Blocks are distributed to each peer for independent validation and replicated storage. The key challenge is to achieve a consensus on the validity and order of transactions and blocks, despite peers that are characterized by "byzantine faults": they may not respond correctly, may respond unpredictably, or may become altogether unresponsive. Additionally, malicious nodes may attempt to undermine the integrity of the chain.

3.1 Proof-of-Work Consensus

Bitcoin popularized the proof-of-work mechanism for consensus finding and securing the blockchain. New transactions are distributed to all peers, validated and added to a transaction pool. Validation is based on transactions that exist in the chain as well as others already in the transaction pool. Each peer can independently propose new blocks based on its latest block and distribute these to other peers. Depending on network connectivity, speeds, and topology, each peer may have a different set of blocks and transactions, and hence may propose different blocks. These may be based on different previous blocks, may contain different transactions, or differ in some minor way, e.g. a different transaction order or creation timestamp. Thus, there may be different blocks referring to the same previous block, leading to *side branches*. Each peer considers the longest branch as the current main branch and proposes new blocks based on this. Transactions in side branches are not considered valid and are not considered when validating new transitions or blocks. When a side branch becomes longer than the current main branch, the chain undergoes a *reorganization*. What was the side branch is validated and becomes the main branch. What was the main branch is considered invalid and becomes a side branch. Transactions no longer in the main branch are added back to the transaction pool to be included in other blocks. As a consequence, different peers can at times consider different blocks and transactions as valid. As proposed blocks are distributed across the network, peers will eventually converge on a consensus regarding the valid blocks and transactions and their order in the main branch of the chain.

To limit the rate of new block proposals and to secure the blockchain against atttacks, proof-of-work consensus requires block proposers to solve a hard problem ("proof-of-work", "mining"). Typically, this is to require the block hash to be less than a certain value. A limited block rate allows nodes to achieve eventual consensus, and a hard problem prevents attackers from "overtaking" the creation of legitimate blocks with fraudulent one. Assuming equal processing power for each node, the network needs $2f + 1$ total nodes to tolerate f faulty or malicious nodes.

The probability that a transaction in the main branch of the blockchain becomes invalid decreases with each block that is "mined" on top of it, although in principle it is always possible that a block is invalidated. Blockchain communities use rules of thumb for the number of additional blocks that is considered to make a transaction "safe" enough to act on, for example, six for Bitcoin and twelve for Ethereum. In addition to the lack of finality of consensus, this approach induces significant latency as applications must wait not only for one block but many to be created. Applications must actively monitor the status of all transactions of interest and must react to chain reorganizations.

3.2 BFT-Based Consensus and State Machine Replication

In response to the drawbacks of the proof-of-work consensus, i.e. latencies, no finality of consensus, and required processing power, proven correct ordering algorithms, based on distributed systems research, have seen a resurgence in interest. Most of the ongoing research can be traced back to a practical method for achieving byzantine fault tolerance (PBFT) [4]. PBFT orders client requests using a set of nodes that are fully connected by reliable messaging. Every ordering consensus is established by a specific set of nodes ("view"), with a leader or primary node. Tolerating up to f faulty nodes requires $3f + 1$ total nodes.

Protocol BPFT is a three-stage protocol. A client sends a request to all nodes. The leader proposes a sequence number for the request and broadcasts a pre-prepare message. Upon receipt of a pre-prepare message, a node broadcasts a corresponding prepare messge if it has itself received the request, has not already received another pre-prepare message for the same sequence number, and is in the current view. This indicates the node is prepared to accept the proposed sequence number. Nodes then wait to receive $2f$ matching prepare messages, indicating that $2f+1$ nodes are prepared to accept the proposed sequence number for the request. When a node has received $2f$ prepare messages, it broadcasts a commit message to all nodes. Each node then waits to reeive $2f$ commit messages, indicating that $2f+1$ nodes have accepted the proposed sequence number for the request. Upon committing, the node executes the request and sends the reply to the client. The client in turn waits for $2f + 1$ replies, which indicates that a consensus has been reached on the sequence number of the request.

In case the leader fails to propose a sequence number, nodes first forward requests to the leader. When the primary continues failing to act on requests or proposes sequence numbers too high or too low, nodes trigger a view change. The view change uses a three-stage protocol similar to the normal operation one to determine a new leader.

Consensus about request sequencing is closely related to state machine replication (SMR). Each node maintains a state that can be changed by client requests. When every node begins with the same state and executes requests in the same order, the state machine is replicated. MOD-SMART [15] is a modularized system for state machine replication that is independent of the underlying BFT consensus mechanism and optimal in the number of required messages. To allow nodes to join a view requires a way to transfer state. Because checkpointing state information can disrupt normal operation, nodes create a checkpoint at different times ("sequential checkpointing"). However, the lack of multiple identical checkpoints means that a simple quorum protocol cannot be used to transfer state to a new node. Instead, the "collaborative state transfer" [2] protocol provides checkpoint and log information from multiple nodes in a way that allows a new node to verify its correctness.

BFT SMART BFT-SMART [3] is a software library built around MOD-SMART, collaborative state transfer and view reconfiguration. It can be configured to provide crash tolerance only, rather than byzantine fault tolerance, significantly increasing its performance. Adding digital signatures to messages allows the system to also tolerate malicious nodes.

The BFT-SMART library provides a simple programming interface. The client-side interface exposes the ability to submit requests for ordered operations or unordered operations. Generally, state-changing operations should be ordered, while read-only operations may be unordered, depending on application requirements. Applications implement a server-side interface (encapsulating the state machine) that receives ordered and unordered operations in consensus sequence from the BFT-SMART library for execution. Any replies are sent back to the requesting client. Operation requests are opaque to the library and are simple byte arrays. It is the client- and server-side application's responsibility to serialize and deserialize these in a meaningul way.

For state management, the server-side application implements methods for the library to fetch and set a state snapshot or checkpoint, which is also serialized as a byte array. State changes (ordered operations) are logged and the state is periodically checkpointed. When a node joins a view, it is sent the latest checkpointed state, which it sets for the server-side application, and any ordered operations after that checkpoint are then replayed as normal operations, allowing the server state to catch up to the consensus state.

View reconfigurations (adding or removing a node, or changing the level of fault tolerance) are special types of ordered requests but are treated as any other ordered request for ordering and consensus purposes.

Summary PBFT-derived ordering, as implemented in BFT-SMART, avoids the latency, lack of finality and processing requirements of proof-of-work consensus. On the other hand, its three-stage protocol imposes significant communication overhead and requires fully-connected nodes. Additionally, proof-of-work consensus guards against a higher proportion of faulty nodes (1/2 versus 1/3 for PBFT-derived consensus). Fault tolerance in PBFT-derived methods increases linearly with the number of nodes, but performance tends to decrease due to additional communication.

4 Architectural Design

The main component of a WfMS is the workflow engine, which interprets the workflow model and enables work items for manual execution or execution by external applications [10]. The engine maintains workflow state information and case data. It may be supported by, or include, services for organizational data management and role resolution, worklist management, document storage, etc. Designing a WfMS architecture requires choosing where to locate and how to implement the workflow engine and other service.

Existing work on blockchain-based workflow management (Sec. 2) has deployed the workflow engine on the blockchain itself. However, by compiling a workflow model to a smart contract, the contract forms a workflow engine for only this workflow model. Alternatively, blockchains can be treated as a trusted infrastructure layer for generic workflow engines, only sharing the state of work and achieving consensus on that state. To our knowledge, there has been no such implementation using PBFT-derived ordering mechanisms.

Ordering, block management, and the workflow engine are the three main services in our system architecture.

Ordering Service The ordering service in our prototype is implemented based on the BFT-SMART library [3]. It can receive transactions from the workflow engine, which is the only ordered (state-changing) type of request it supports. The ordering service maintains a record of the latest block hash and block number, as well as a queue of transactions that have been added as its state. When a sufficient number of transactions has been collected, the ordering service creates a new block and clears the transaction queue. Clients can request the latest block hash; this is an unordered type of request.

Block Service The block service stores the blockchain, may exchange blocks with other nodes, and verifies the integrity of the blockchain.

The block service uses a peer-to-peer network for block exchange with new and recovering peers. This network is distinct from the network layer of BFT-SMART and is not fully connected. Block exchange is required only when a node begins operation and enters an ordering view. At that point, the ordering service state is first updated through the BFT-SMART state replication mechanisms. The block service then compares its latest block to the latest hash from the ordering service. The latter is assumed to be authoritative. Verification of the blockchain then proceeds backwards from the head of the chain, i.e. the block with the latest hash. Any missing blocks are requested from other peers and verified prior to adding them.

Workflow Engine The engine is notified by the block service when a new block is added to the chain. It then executes all transactions in the block, updating the state of each process instance and creating work items accordingly. It manages user interactions with work items and execution of external functions by work items. Upon work item completion, the engine generates a new transaction and passes it to the ordering service.

Next, we discuss architectural options that we were presented with when designing our prototype system. These affect performance, ease of implementation, and resilience.

4.1 BFT SMR State

Because BFT-SMART provides a very high abstraction for checkpointing, logging and exchanging state information with new and recovering nodes one architectural option is to employ this method also for the blocks of the blockchain. This means that the entire blockchain is part of the replicated state in

BFT-SMART, effectively removing the need for a separate block service with its peer-to-peer network and block exchange protocol. While easy to implement by serializing the blockchain into the BFT-SMART byte array snapshot, this model becomes infeasible as the blockchain becomes too large to rapidly exchange with other nodes using the complex and communication-intensive state transfer mechanism in BFT-SMART. Instead, it is sufficient for the state to only contain the hash of the last block, the number of the last created block, and the queue of transactions waiting to be collected into new blocks.

4.2 Block Creation

As noted above, blocks are created by the ordering service. One design option is to pass new blocks as replies from the ordering service operation back to the node that requested to the add-transition operation that triggered the block creation. That node's block service is then responsible for exchanging the block with other nodes using the peer-to-peer network. This creates significant traffic on that network and may also lead to delays in new block distribution.

A second design option is to have the ordering service server-side application that creates the new block pass the new block directly to the block service on that node. This tighter coupling between ordering service and block service reduces the communication overhead for the peer-to-peer network and latencies due to the block exchange. The peer-to-peer network is still required for block exchange with new or recovering nodes.

4.3 Coupling of Block Service and Workflow Engine

One option is for workflow engine and block service to always be present together on each node. Block service notifying the engine of new blocks, or the engine validating transactions for the block service can be done with simple and fast method calls.

While there is little to be gained by separating block service and workflow engine and running multiples of each, a second option is to operate only a single block service with multiple, distributed workflow engines. This eliminates the peer-to-peer network and block exchange communication. Blockchain integrity can still be verified from the latest hash of the ordering service nodes. However, this design eliminates the redundancy that is an advantage of a replicated blockchain. On the other hand, redundancy can be achieved by a replicated storage layer with the block service, e.g. a distributed file system.

4.4 Workflow State or Workflow Operations

First, a transaction may represent workflow operations such as defining a new workflow model, launching a new case, executing an activity, aborting or cancelling a case or removing a workflow model. Activity execution information includes the activity name and case ID, as well input and output data values.

Alternatively, a transaction can represent a workflow instance state, i.e. data values and enabled activities, without capturing activity execution itself. Including both is inefficient and the redundancy may threaten consistency.

The first option requires the engine to maintain its own state of the workflow (i.e. information about workflow models, running instances, data values and enabled activities). Constructing this state means reading the blockchain forwards from the genesis block and replaying all transactions. State updates are done by executing transactions in new blocks. While reducing the amount of information stored on the blockchain, this option requires significant effort in managing the separate state and ensuring it is consistent with the blockchain record. The second option makes the workflow state available by reading the blockchain backwards from the head to identify the latest state for each process instance. State updates are done simply by copying workflow states from transactions as new blocks are presented. Not maintaining separate state signifanctly simplifies the workflow engine design.

The first option provides activity information in each transaction. Hence, data constraints can be specified as post-exeuction constraints and checked when validating the transaction. The second option does not provide information about activity execution in a transaction. Hence, only global case data constraints can be specified and checked as part of transaction validation.

Finally, while transactions are waiting to be included in a block, users can be made aware of such pending transactions. For the first option, transactions are informative as they inform the user about activities. In the second option, such transactions are less informative to the user, as they do not contain specific activity execution information.

4.5 Block Size

In proof-of-work blockchains, blocks contain multiple transactions. The block size is a trade-off among transaction arrival rate, available hashing power, desired block arrival rate, available network bandwidth, and tolerance for latency. A transaction may be ”pending” for a some time until it is included in a block and at a ”safe” depth. In contrast, in PBFT-based systems, there is nothing to prevent blocks from containing only one transaction, i.e. the blockchain becomes a chain of individual transactions.

Proof-of-work systems order transactions between different blocks, but the order of transactions within a block is not defined: Transactions may be included in a block as long as they are not mutually contradictory. Block miners ultimately impose an order, but this order is arbitrary. This means that as pending transactions are collected, they must be validated against the *entire* set of pending transactions to ensure they are not mutually conflicting. In contrast, when a client in a BFT system requests that a new transaction be added to the pool, this request is totally ordered by the BFT algorithm and the transaction must be validated only against the *immediately prior* one.

4.6 Coupling of Block Service and Ordering Service

The ordering and block services (the latter always together with a workflow engine), can be coupled or integrated to varying degrees. At one extreme, block management is part of the ordering service, as discussed in Sec. 4.1.

In a slightly less integrated architecture, every block service and workflow engine node is also an ordering node and vice versa, but block management is distinct from ordering and implements its own peer-to-peer network infrastructure. This allows each ordering node to quickly validate transactions using the local workflow engine. The drawback of this three-in-one integrated-node design is that the number of ordering nodes should be determined by the desired level of fault tolerance, whereas the number of workflow nodes should be determined based on the business process and/or application. An application requiring more ordering than workflow nodes is not a problem as the additional nodes are simply not assigned any workflow tasks. On the other hand, when an application requires more workflow nodes than ordering nodes, the excess ordering nodes decrease performance due to the communication overhead.

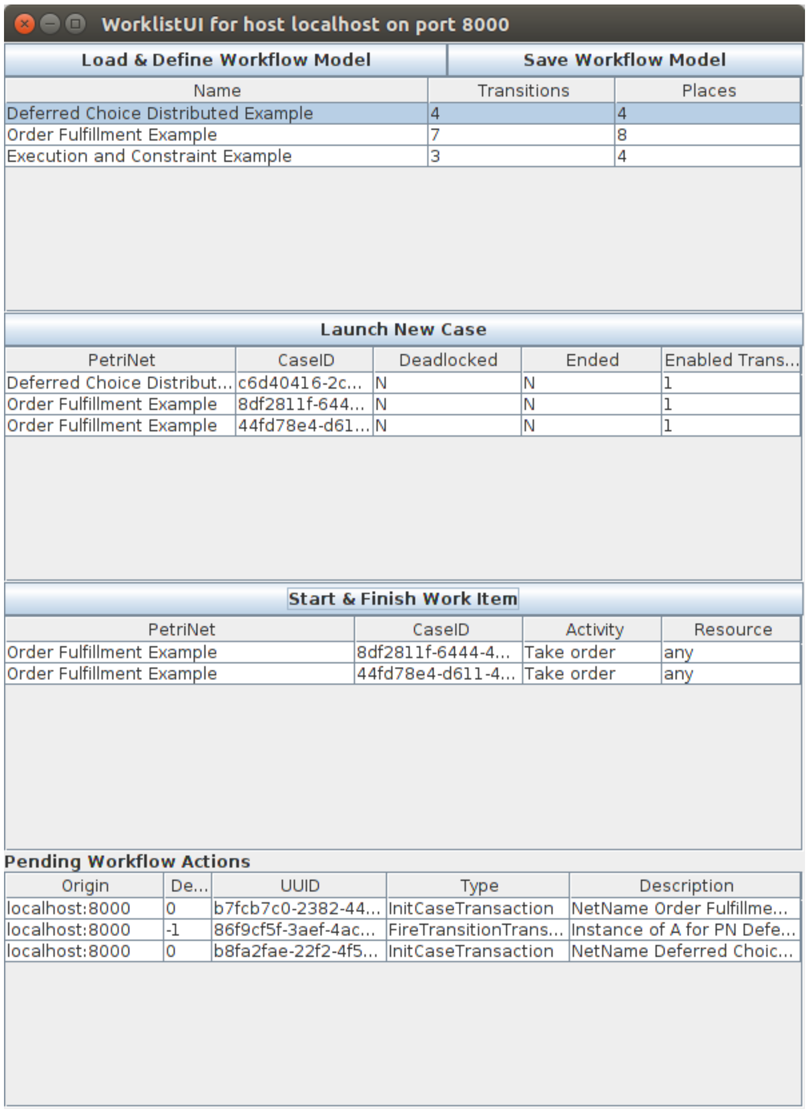
In a very loosely coupled architecture, ordering nodes and block service/ workflow nodes are separated. Newly created blocks are passed to the block server as replies from BFT operations and are communicated using the block service peer-to-peer network. However, because the ordering service validates transactions after ordering but before accepting them, each ordering node would require a reliable connection to at least one workflow engine. Managing these connections as workflow engines join and leave the network, and managing the additional communication, adds significant complexity and introduces additional latency in validating transactions.

5 Prototype Implementation

Given the architectural design options discussed in the previous section and their advantages and disadvantages, we chose to implement our initial prototype by storing only the latest block hash, block number, and transaction pool as BFT-SMART state (Sec. 4.1). The workflow engine and block service are always present together at each node (Sec. 4.3) and both are always co-located with an ordering service node (Sec. 4.6). The ordering service passes new blocks directly to the local block service upon block creation, but a peer-to-peer network supports block exchange with new or recovering nodes (Sec. 4.2). We store workflow states on the blockchain, instead of workflow operations (Sec. 4.4) so as not having to maintain a separate workflow state in the engine. The block size is user configurable (Sec. 4.5). We developed the prototype in Java. Source code is available[3]. Fig. 1 shows a screenshot of our prototype.

We implemented a private peer-to-peer infrastructure with a pre-defined list of participating actors. To keep our prototype simple, actors are identified by their internet address rather than their public keys, so that we can omit an

[3] https://joerg.evermann.ca/software.html


WorklistUI for host localhost on port 8000
Load & Define Workflow Model
Save Workflow Model
Launch New Case
Start & Finish Work Item

Name	Transitions	Places
Deferred Choice Distributed Example	4	4
Order Fulfillment Example	7	8
Execution and Constraint Example	3	4

PetriNet	CaseID	Deadlocked	Ended	Enabled Trans...
Deferred Choice Distribut...	c6d40416-2c...	N	N	1
Order Fulfillment Example	8df2811f-644...	N	N	1
Order Fulfillment Example	44fd78e4-d61...	N	N	1

PetriNet	CaseID	Activity	Resource
Order Fulfillment Example	8df2811f-6444-4...	Take order	any
Order Fulfillment Example	44fd78e4-d611-4...	Take order	any

Pending Workflow Actions

Origin	De...	UUID	Type	Description
localhost:8000	0	b7fcb7c0-2382-44...	InitCaseTransaction	NetName Order Fulfillme...
localhost:8000	-1	86f9cf5f-3aef-4ac...	FireTransitionTrans...	Instance of A for PN Defe...
localhost:8000	0	b8fa2fae-22f2-4f5...	InitCaseTransaction	NetName Deferred Choic...

Fig. 1. Screenshot of prototype

address resolution layer. The P2P layer is implemented using Java sockets and serialization. Each P2P node has an outbound server that establishes connections to other peers, and an inbound server that accepts and verifies connection requests from peers. 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. Nodes can join and leave the peer-to-peer network at will. When a node joins, it tries to open connections to running peers. The first peer to be contacted will initiate a view change in the BFT-SMART odering service to include the new peer on that level as well.

Upon starting of a node, the BFT-SMART layer will first update state information from other nodes in the view. Next, the block service will identify missing blocks and request them from peers. Once the blockchain is complete and verified, the workflow engine reads the blockchain to get the latest state for each workflow instance. Peer-to-peer messages are cryptographically signed and verified upon receipt. Table 1 lists the message types on our peer-to-peer network.

BlockRequest	Requests a block with a specific hash from one or more peers
BlockSend	Sends a block to one or more peers
BlockChainRequest	Requests multiple blocks within a hash range from one or more peers
BlockChainSend	Sends multiple blocks to one or more peers

Table 1. Message types

Our blockchain has two transaction types. A *ModelUpdate* transaction installs a new workflow model definition. An *InstanceState* transaction contains a state of a workflow instance. It is submitted after a new case has been launched or an activity instance has been executed. Extensions to cancel cases and uninstall model definitions are readily possible.

To keep our prototype simple, our workflow models are based on plain Petri nets [1]. Each Petri net transition specifies a workflow activity. The workflow engine keeps track of the Petri net markings and case data, and can detect deadlocked and finished cases to remove them from the worklist.

Each activity is associated with a single node. This partitioning of the process to different nodes does not form the resource perspective of the workflow but is used only to signal each node whether to act on a transaction. Each node can provide its own resource management by defining roles or other organizational concepts and performing further work item allocation within each node. Our models allow the process designer to specify this information. External method calls are specified as calls to static Java methods, and are performed synchronously by the workflow engine on work item enablement.

The data perspective is implemented as a key–value store. We currently admit only simple Java types as we implement a GUI for these; an extension to arbitrary types is readily possible. Each workflow instance has a set of data variables. When a transition is enabled, an activity instance (work item) is created for it and its input values are filled from the values of the workflow instance. The activity instance is then added to the local worklist or externally executed. After an activity instance is completed (manually or through execution of an external application), output values are written back to the workflow instance.

The ordering service, workflow engine and the block service have a simple interface (Table 2). The ordering and block services can call on the workflow engine to validate transactions against the current workflow state, and optionally, against the most recent pending transaction. Validation checks that a transaction's instance marking is reachable from the marking of the current workflow instance state or that of the pending transaction. It also checks for data constraint violation. The block service receives new blocks from the ordering service and passes them to the workflow engine. In the other direction, the workflow engine can submit new transactions to the ordering service after a work item has been completed. Finally, the block service can request the latest hash from the ordering service on joining the network or recovering from a fault.

→	validateTransaction(tx[, pendingTx])	Ordering service asks workflow engine to validate a transaction, given the current workflow state and optionally the most recent pending transaction
→	receiveBlock(block)	Block service receives a new block and passes relevant transactions to the workflow engine
←	addTransaction(tx)	Workflow engine submits a new transaction to the ordering service
←	getLatestHash()	Block service requests the latest hash from the ordering service

Table 2. Interfaces between ordering service, block service and workflow engine (directions from the perspective of the ordering service)

6 Discussion and Conclusions

Previous work on blockchain-based WfMS has focused on creating smart contracts to represent specific workflow models. In particular, the Ethereum proof-of-work-based blockchain is widely used. However, proof-of-work-based systems have significant drawbacks in terms of processing power requirements, latency, and the lack of final consensus. In this work, we have shown that a PBFT-derived ordering and consensus method is a suitable WfMS infrastructure.

Through the development of our prototype, we have identified architectural design options with their advantages and disadvantages. Our chosen design, in which we integrate ordering service, block service, and workflow engine on every node, strikes a balance between architectural and implementation simplicity on the one hand, and performance and scalability on the other.

A limitation in our chosen model is that the number of nodes must strike a balance between the requirements of the workflow (the number of actors involved), the desired level of fault tolerance, and the performance of the system. The major advantages are the low communication overhead on the P2P block exchange and the ability of local workflow engines to validate transactions quickly.

While our approach has lower resilience against faults and malicious attacks than proof-of-work chains, it also has lower latency and higher throughput. Unlike proof-of-work chains, the PBFT-based approach does not scale to a very large number of nodes. Given these characteristics, systems such as ours are suitable for private blockchain applications where extreme levels of maliciousness are unlikely. The low latency makes them suitable for fast-moving processes, where activities are of short duration and must follow each other quickly. Our system is cheaper to operate than public proof-of-work blockchains that incentivizes block mining through cryptocurrencies. While one can implement private proof-of-work chains, these lose their resilience against attacks in small networks as it is easy for a single actor to acquire the majority of processing power in a single high-performance node. To attack a PBFT-derived system requires control of more than 1/3 of all nodes, which is more difficult to achieve, especially in the absence of trust among actors.

From the user's perspective, our system is little different from traditional WfMS. Because transactions need not wait to be included in the blockchain or to be mined to a "safe" depth and consensus is final, latency is low and the execution status of workflow activities cannot change and does not need to be monitored and reported to the user.

Our work on the prototype implementation has shown some avenues for future research.

- We currently assign single peer nodes statically to workflow activities. In the future, we hope to extend this to dynamic peer node assignment and to integrate this with the workflow's resource perspective.
- Porting existing workflow engines, such as the open-source YAWL system [9], to blockchain infrastructure allows a richer workflow language and leverages existing implementations.

To conclude, this paper has described a prototype implementation for an architecture that has not yet seen any attention in the blockchain-based workflow literature. We have implemented PBFT-based system as recommended by [16] and shown that this infrastructure is suitable for WfMS. We have shown how generic workflow engines can be readily adapted to fit onto a blockchain infrastructure without implementing these as smart contracts. The interfaces between components are quite simple. In contrast to [13], who suggest that blockchain-specific modelling languages need to be developed, our work shows that workflow engines do not need to be implemented using smart contracts, as done by [17, 18], but that traditional workflow engines be easily adapted to use blockchains as infrastructure for communication, persistence, replication, and trust building.

References

1. van der Aalst, W.M.P.: The application of petri nets to workflow management. Journal of Circuits, Systems, and Computers **8**(1), 21–66 (1998).
2. Bessani, A.N., Santos, M., Felix, J., Neves, N.F., Correia, M.: On the efficiency of durable state machine replication. In: Birrell, A., Sirer, E.G. (eds.) 2013 USENIX Annual Technical Conference, San Jose, CA, USA, June 26-28, 2013. pp. 169–180. USENIX Association (2013)

3. Bessani, A.N., Sousa, J., Alchieri, E.A.P.: State machine replication for the masses with BFT-SMART. In: 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2014, Atlanta, GA, USA, June 23-26, 2014. pp. 355–362. IEEE Computer Society (2014)
4. Castro, M., Liskov, B.: Practical byzantine fault tolerance and proactive recovery. ACM Trans. Comput. Syst. **20**(4), 398–461 (2002)
5. Fridgen, G., Sablowsky, B., Urbach, N.: Implementation of a blockchain workflow management prototype. ERCIM News **2017**(110) (2017)
6. Fridgen, G., Radszuwill, S., Urbach, N., Utz, L.: Cross-organizational workflow management using blockchain technology - towards applicability, auditability, and automation. In: 51st Hawaii International Conference on System Sciences HICSS. AIS Electronic Library (2018)
7. García-Bañuelos, L., Ponomarev, A., Dumas, M., Weber, I.: Optimized execution of business processes on blockchain. In: Carmona, J., Engels, G., Kumar, A. (eds.) Business Process Management - 15th International Conference, BPM, Proceedings. Lecture Notes in Computer Science, vol. 10445, pp. 130–146. Springer (2017).
8. Härer, F.: Decentralized business process modeling and instance tracking secured by a blockchain. In: Bednar, P.M., Frank, U., Kautz, K. (eds.) 26th European Conference on Information Systems ECIS. p. 55. AIS Electronic Library (2018)
9. ter Hofstede, A.H.M., van der Aalst, W.M.P., Adams, M., Russell, N. (eds.): Modern Business Process Automation. Springer (2010)
10. Hollingsworth, D.: The workflow reference model. Tech. rep., Workflow Management Coalition (1995)
11. Hukkinen, T., Mattila, J., Seppälä, T., et al.: Distributed workflow management with smart contracts. Tech. rep., Research Institute Finnish Economy (2017)
12. López-Pintado, O., García-Bañuelos, L., Dumas, M., Weber, I.: 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., Weske, M. (eds.) Proceedings of the BPM Demo Track co-located with 15th International Conference on Business Process Modeling. CEUR vol. 1920 (2017)
13. Mendling, J., Weber, I., van der Aalst, W.M.P., vom Brocke, J., Cabanillas, C., et al.: Blockchains for business process management - challenges and opportunities. ACM Trans. Management Inf. Syst. **9**(1), 4:1–4:16 (2018).
14. Rimba, P., Tran, A.B., Weber, I., Staples, M., Ponomarev, A., Xu, X.: Comparing blockchain and cloud services for business process execution. In: 2017 IEEE International Conference on Software Architecture, ICSA. pp. 257–260. IEEE Computer Society (2017).
15. Sousa, J., Bessani, A.N.: From byzantine consensus to BFT state machine replication: A latency-optimal transformation. In: Constantinescu, C., Correia, M.P. (eds.) 2012 Ninth European Dependable Computing Conference, Sibiu, Romania, May 8-11, 2012. pp. 37–48. IEEE Computer Society (2012).
16. Viriyasitavat, W., Hoonsopon, D.: Blockchain characteristics and consensus in modern business processes. Journal of Industrial Information Integration (2018)
17. Weber, I., Xu, X., Riveret, R., Governatori, G., Ponomarev, A., Mendling, J.: Using blockchain to enable untrusted business process monitoring and execution. Tech. rep., Technical Report CSE-TR-201609, University of New South Wales (2016)
18. Weber, I., Xu, X., Riveret, R., Governatori, G., Ponomarev, A., Mendling, J.: Untrusted business process monitoring and execution using blockchain. In: Rosa, M.L., Loos, P., Pastor, O. (eds.) Business Process Management - 14th International Conference, BPM, Proceedings. Lecture Notes in Computer Science, vol. 9850, pp. 329–347. Springer (2016).