Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains

Elli Androulaki Artem Barger Vita Bortnikov Christian Cachin Konstantinos Christidis Angelo De Caro David Enyeart Christopher Ferris Gennady Laventman Yacov Manevich Srinivasan Muralidharan * Chet Murthy † Binh Nguyen * Manish Sethi Gari Singh Keith Smith Alessandro Sorniotti Chrysoula Stathakopoulou Marko Vukolić Sharon Weed Cocco Jason Yellick

I B M

Abstract

Hyperledger Fabric is a modular and extensible open-source system for deploying and operating permissioned blockchains. Fabric is currently used in more than 400 prototypes and proofs-of-concept of distributed ledger technology, as well as several production systems, across different industries and use cases.

Starting from the premise that there are no "one-size-fits-all" solutions, Fabric is the first truly extensible blockchain system for running distributed applications. It supports modular consensus protocols, which allows the system to be tailored to particular use cases and trust models. Fabric is also the first blockchain system that runs distributed applications written in general-purpose programming languages, without systemic dependency on a native cryptocurrency. This stands in sharp contrast to existing blockchain platforms for running smart contracts that require code to be written in domain-specific languages or rely on a cryptocurrency. Furthermore, it uses a portable notion of membership for realizing the permissioned model, which may be integrated with industry-standard identity management. To support such flexibility, Fabric takes a novel approach to the design of a permissioned blockchain and revamps the way blockchains cope with non-determinism, resource exhaustion, and performance attacks.

This paper describes Fabric, its architecture, the rationale behind various design decisions, its security model and guarantees, its most prominent implementation aspects, as well as its distributed application programming model. We further evaluate Fabric by implementing and benchmarking a Bitcoin-inspired digital currency. We show that Fabric achieves end-to-end throughput of more than 3500 transactions per second in certain popular deployment configurations, with sub-second latency.

*Work done at IBM, now with State Street Corp.

†Work done at IBM.

1. Introduction

A blockchain can be defined as an immutable *ledger* for recording *transactions*, maintained within a distributed network of mutually untrusting *peers*. Every peer maintains a copy of the ledger. The peers execute a *consensus protocol* to validate transactions, group them into blocks, and build a hash chain over the blocks. This process forms the ledger by ordering the transactions, as is necessary for consistency. Blockchains have emerged with Bitcoin (`http://bitcoin.org/`) and are widely regarded as a promising technology to run trusted exchanges in the digital world.

In a *public* or *permissionless* blockchain anyone can participate without a specific identity. Public blockchains typically involve a native cryptocurrency and often use consensus based on "proof of work" (PoW) and economic incentives. *Permissioned* blockchains, on the other hand, run a blockchain among a set of known, identified participants. A permissioned blockchain provides a way to secure the interactions among a group of entities that have a common goal but which do not fully trust each other, such as businesses that exchange funds, goods, or information. By relying on the identities of the peers, a permissioned blockchain can use traditional Byzantine-fault tolerant (BFT) consensus.

Blockchains may execute arbitrary, programmable transaction logic in the form of *smart contracts*, as exemplified by Ethereum (`http://ethereum.org/`). The scripts in Bitcoin were a predecessor of the concept. A smart contract functions as a *trusted distributed application* and gains its security from the blockchain and the underlying consensus among the peers. This closely resembles the well-known approach of building resilient applications with state-machine replication (SMR) [31]. However, blockchains depart from traditional SMR with Byzantine faults in important ways: (1)

not only one, but many distributed applications run concurrently; (2) applications may be deployed dynamically and by anyone; and (3) the application code is untrusted, potentially even malicious. These differences necessitate new designs.

Many existing smart-contract blockchains follow the blueprint of SMR [31] and implement so-called *active* replication [13]: a protocol for *consensus* or *atomic broadcast* first orders the transactions and propagates them to all peers; and second, each peer executes the transactions sequentially. We call this the *order-execute architecture*; it requires all peers to execute every transaction and all transactions to be deterministic. The order-execute architecture can be found in virtually all existing blockchain systems, ranging from public ones such as Ethereum (with PoW-based consensus) to permissioned ones (with BFT-type consensus) such as Tendermint (`http://tendermint.com/`), Chain (`http://chain.com/`), and Quorum (`http://www.jpmorgan.com/global/Quorum`). Although the order-execute design is not immediately apparent in all systems, because the additional transaction validation step may blur it, the limitations of order-execute are inherent in all: every peer executes every transaction and transactions must be deterministic.

Prior permissioned blockchains suffer from many limitations, which often stem from their permissionless relatives or from using the order-execute architecture. In particular:

- Consensus is *hard-coded* within the platform, which contradicts the well-established understanding that there is no "one-size-fits-all" (BFT) consensus protocol [33];
- The *trust model* of transaction validation is determined by the consensus protocol and cannot be adapted to the requirements of the smart contract;
- Smart contracts must be written in a *fixed, non-standard, or domain-specific language*, which hinders wide-spread adoption and may lead to programming errors;
- The *sequential execution* of all transactions by all peers *limits performance*, and complex measures are needed to prevent denial-of-service attacks against the platform originating from untrusted contracts (such as accounting for runtime with "gas" in Ethereum);
- Transactions must be *deterministic*, which can be difficult to ensure programmatically;
- Every smart contract runs on *all* peers, which is at odds with *confidentiality*, and prohibits the dissemination of contract code and state to a subset of peers.

In this paper we describe *Hyperledger Fabric* or simply *Fabric*, an open-source (`http://github.com/hyperledger/fabric`) blockchain platform that overcomes these limitations. Fabric is one of the projects of Hyperledger (`http://www.hyperledger.org`) under the auspices of the Linux Foundation (`http://www.linuxfoundation.org`). Fabric is used in more than 400 prototypes, proofs-of-concept, and in production distributed-ledger systems, across different industries and use cases. These use cases include but are not limited to areas such as dispute resolution, trade logistics, FX netting, food safety, contract management, diamond provenance, rewards point management, low liquidity securities trading and settlement, identity management, and settlement through digital currency.

Fabric introduces a new blockchain architecture aiming at resiliency, flexibility, scalability, and confidentiality. Designed as a modular and extensible general-purpose permissioned blockchain, Fabric supports the execution of distributed applications written in standard programming languages. This makes Fabric the first *distributed operating system* for permissioned blockchains.

The architecture of Fabric follows a novel *execute-order-validate* paradigm for distributed execution of untrusted code in an untrusted environment. It separates the transaction flow into three steps, which may be run on different entities in the system: (1) *executing* a transaction and checking its correctness, thereby *endorsing* it (corresponding to "transaction validation" in other blockchains); (2) *ordering* through a consensus protocol, irrespective of transaction semantics; and (3) transaction *validation* per application-specific trust assumptions, which also prevents race conditions due to concurrency.

This design departs radically from the order-execute paradigm in that Fabric typically executes transactions before reaching final agreement on their order. It combines the two well-known approaches to replication, *passive* and *active*, as follows.

First, Fabric uses *passive* or *primary-backup replication* [6, 13] as often found in distributed databases, but with middleware-based asymmetric update processing [24, 25] and ported to untrusted environments with Byzantine faults. In Fabric, every transaction is executed (endorsed) only by a subset of the peers, which allows for parallel execution and addresses potential non-determinism, drawing on "execute-verify" BFT replication [21]. A flexible endorsement policy specifies which peers, or how many of them, need to vouch for the correct execution of a given smart contract.

Second, Fabric incorporates *active replication* in the sense that the transaction's effects on the ledger state are only written after reaching consensus on a total order among them, in the deterministic validation step executed by each peer individually. This allows Fabric to respect application-specific trust assumptions according to the transaction endorsement. Moreover, the ordering of state updates is delegated to a modular component for consensus (i.e., atomic broadcast), which is stateless and logically decoupled from the peers that execute transactions and maintain the ledger. Since consensus is modular, its implementation can be tailored to the trust assumption of a particular deployment. Although it is readily possible to use the blockchain peers also for implementing consensus, the separation of the two roles adds flexibility and allows one to rely on well-established toolkits for CFT (crash fault-tolerant) or BFT ordering.

Overall, this *hybrid replication* design, which mixes passive and active replication in the Byzantine model, and the *execute-order-validate* paradigm, represent the main innovation in Fabric architecture. They resolve the issues mentioned before and make Fabric a scalable system for permissioned blockchains supporting flexible trust assumptions.

To implement this architecture, Fabric contains modular building blocks for each of the following components:

Ordering service: An *ordering service* atomically broadcasts state updates to peers and establishes consensus on the order of transactions. It has been implemented with Apache Kafka/ZooKeeper (`http://kafka.apache.org/`) and with BFT-SMaRt [3].

Identity and membership: A *membership service provider* is responsible for associating peers with cryptographic identities. It maintains the permissioned nature of Fabric.

Scalable dissemination: An optional *peer-to-peer gossip service* disseminates the blocks output by ordering service to all peers.

Smart-contract execution: Smart contracts in Fabric run within a container environment for isolation. They can be written in standard programming languages but do not have direct access to the ledger state.

Ledger maintenance: Each peer locally maintains the ledger in the form of the append-only blockchain and as a snapshot of the most recent state in a *key-value store (KVS)*. The KVS can be implemented by standard libraries, such as LevelDB or Apache CouchDB.

The remainder of this paper describes the architecture of Fabric and our experience with it. Section 2 summarizes the state of the art and explains the rationale behind various design decisions. Section 3 introduces the architecture and the execute-order-validate approach of Fabric in detail, illustrating the transaction execution flow. In Section 4, the key components of Fabric are defined, in particular, the ordering service, membership service, peer-to-peer gossip, ledger database, and smart-contract API. Results and insights gained in a performance evaluation of Fabric with a Bitcoin-inspired cryptocurrency, deployed in a cluster environment on commodity public cloud VMs, are given in Section 5. They show that Fabric achieves, in popular deployment configurations, throughput of more than 3500 tps, achieving finality [36] with latency of a few hundred ms. Finally, Section 6 discusses related work.

2. Background

2.1 Order-Execute Architecture for Blockchains

All previous blockchain systems, permissioned or not, follow the order-execute architecture. This means that the blockchain network orders transactions first, using a consensus protocol, and then executes them in the same order on all peers sequentially.[1]

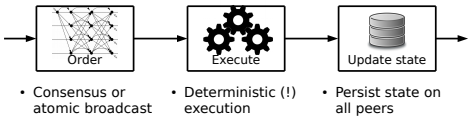

Figure 1. Order-execute architecture in replicated services.

For instance, a PoW-based permissionless blockchain such as Ethereum combines consensus and execution of transactions as follows: (1) every peer (i.e., a node that participates in consensus) assembles a block containing valid transactions (to establish validity, this peer already pre-executes those transactions); (2) the peer tries to solve a PoW puzzle [28]; (3) if the peer is lucky and solves the puzzle, it disseminates the block to the network via a gossip protocol; and (4) every peer receiving the block validates the solution to the puzzle *and* all transactions in the block. Effectively, every peer thereby repeats the execution of the lucky peer from its first step. Moreover, all peers execute the transactions *sequentially* (within one block and across blocks). The order-execute architecture is illustrated by Fig. 1.

Existing permissioned blockchains such as Tendermint, Chain, or Quorum typically use BFT consensus [9], provided by PBFT [11] or other protocols for atomic broadcast. Nevertheless, they all follow the same order-execute approach and implement classical *active* SMR [13, 31].

2.2 Limitations of Order-Execute

The order-execute architecture is conceptually simple and therefore also widely used. However, it has several drawbacks when used in a general-purpose permissioned blockchain. We discuss the three most significant ones next.

Sequential execution. Executing the transactions sequentially on all peers limits the effective throughput that can be achieved by the blockchain. In particular, since the throughput is inversely proportional to the execution latency, this may become a performance bottleneck for all but the simplest smart contracts. Moreover, recall that in contrast to traditional SMR, the blockchain forms a universal computing engine and its payload applications might be deployed by an adversary. A denial-of-service (DoS) attack, which severely reduces the performance of such a blockchain, could simply introduce smart contracts that take a very long time to execute. For example, a smart contract that executes an infinite loop has a fatal effect, but cannot be detected automatically because the halting problem is unsolvable.

To cope with this issue, public programmable blockchains with a cryptocurrency account for the execution cost. Ethereum, for example, introduces the concept of *gas* con-

[1] In many blockchains with a hard-coded primary application, such as Bitcoin, this transaction execution is called "transaction validation." Here we call this step *transaction execution* to harmonize the terminology.

sumed by a transaction execution, which is converted at a *gas price* to a cost in the cryptocurrency and billed to the submitter of the transaction. Ethereum goes a long way to support this concept, assigns a cost to every low-level computation step, and introduces its own VM monitor to control execution. Although this appears to be a viable solution for public blockchains, it is not adequate for the permissioned model for a general-purpose system without a native cryptocurrency.

The distributed-systems literature proposes many ways to improve performance compared to sequential execution, for instance through parallel execution of unrelated operations [30]. Unfortunately, such techniques are still to be applied successfully in the blockchain context of smart contracts. For instance, one challenge is the requirement for deterministically inferring all dependencies across smart contracts, which is particularly challenging when combined with possible confidentiality constraints. Furthermore, these techniques are of no help against DoS attacks by contract code from untrusted developers.

Non-deterministic code. Another important problem for an order-execute architecture are non-deterministic transactions. Operations executed after consensus in active SMR must be deterministic, or the distributed ledger "forks" and violates the basic premise of a blockchain, that all peers hold the same state. This is usually addressed by programming blockchains in domain-specific languages (e.g., Ethereum Solidity) that are expressive enough for their applications but limited to deterministic execution. However, such languages are difficult to design for the implementer and require additional learning by the programmer. Writing smart contracts in a general-purpose language (e.g., Go, Java, C/C++) instead appears more attractive and accelerates the adoption of blockchain solutions.

Unfortunately, generic languages pose many problems for ensuring deterministic execution. Even if the application developer does not introduce obviously non-deterministic operations, hidden implementation details can have the same devastating effect (e.g., a map iterator is not deterministic in Go). To make matters worse, on a blockchain the burden to create deterministic applications lies on the potentially untrusted programmer. Only one non-deterministic contract created with malicious intent is enough to bring the whole blockchain to a halt. A modular solution to filter diverging operations on a blockchain has also been investigated [8], but it appears costly in practice.

Confidentiality of execution. According to the blueprint of public blockchains, many permissioned systems run all smart contracts on all peers. However, many intended use cases for permissioned blockchains require *confidentiality*, i.e., that access to smart-contract logic, transaction data, or ledger state can be restricted. Although cryptographic techniques, ranging from data encryption to advanced zero-knowledge proofs [2] and verifiable computation [26], can help to achieve confidentiality, this often comes with a considerable overhead and is not viable in practice.

Fortunately, it suffices to *propagate the same state* to all peers instead of running the same code everywhere. Thus, the execution of a smart contract can be restricted to a subset of the peers trusted for this task, that vouch for the results of the execution. This design departs from active replication towards a variant of *passive* replication [6], adapted to the trust model of blockchain.

2.3 Further Limitations of Existing Architectures

Fixed trust model. Most permissioned blockchains rely on asynchronous BFT replication protocols to establish consensus [36]. Such protocols typically rely on a security assumption that among $n > 3f$ peers, up to f are tolerated to misbehave and exhibit so-called *Byzantine faults* [4]. The same peers often execute the applications as well, under the same security assumption (even though one could actually restrict BFT execution to fewer peers [37]). However, such a quantitative trust assumption, irrespective of peers' roles in the system, may not match the trust required for smart-contract execution. In a flexible system, trust at the application level should not be fixed to trust at the protocol level. A general-purpose blockchain should decouple these two assumptions and permit flexible trust models for applications.

Hard-coded consensus. Fabric is the first blockchain system that introduced pluggable consensus. Before Fabric, virtually all blockchain systems, permissioned or not, came with a hard-coded consensus protocol. However, decades of research on consensus protocols have shown there is no such "one-size-fits-all" solution. For instance, BFT protocols differ widely in their performance when deployed in potentially adversarial environments [33]. A protocol with a "chain" communication pattern exhibits provably optimal throughput on a LAN cluster with symmetric and homogeneous links [18], but degrades badly on a wide-area, heterogeneous network. Furthermore, external conditions such as load, network parameters, and actual faults or attacks may vary over time in a given deployment. For these reasons, BFT consensus should be inherently reconfigurable and ideally adapt dynamically to a changing environment [1]. Another important aspect is to match the protocol's trust assumption to a given blockchain deployment scenario. Indeed, one may want to replace BFT consensus with a protocol based on an alternative trust model such as XFT [27], or a CFT protocol, such as Paxos/Raft [29] and ZooKeeper [20], or even a permissionless protocol.

2.4 Experience with Order-Execute Blockchain

Prior to realizing the execute-order-validate architecture of Fabric, the team gained experience with building a permissioned blockchain platform in the order-execute model, with PBFT [11] for consensus. From feedback obtained in many proof-of-concept applications, the limitations of this

approach became immediately clear. For instance, users often observed diverging states at the peers and reported a bug in the consensus protocol; in *all cases*, closer inspection revealed that the culprit was non-deterministic transaction code. Other complaints addressed limited performance, e.g., "only five transactions per second," until users confessed that their average transaction took 200ms to execute. We have learned that the key properties of a blockchain system, namely consistency, security, and performance, must *not* depend on the knowledge and goodwill of its users, in particular since the blockchain should run in an untrusted environment.

3. Architecture

In this section, we introduce the three-phase *execute-order-validate* architecture and then explain the transaction flow. The components of Fabric are discussed in Section 4.

3.1 Fabric Overview

Fabric is a distributed operating system for permissioned blockchains that executes distributed applications written in general-purpose programming languages (e.g., Go, Java, Node.js). It securely tracks its execution history in an append-only replicated ledger data structure and has no cryptocurrency built in.

Fabric introduces the *execute-order-validate* blockchain architecture and does not follow the standard order-execute design, for reasons explained in Section 2. In a nutshell, a distributed application for Fabric consists of two parts:

- A smart contract, called *chaincode*, which is program code that implements the application logic and runs during the *execution phase*. The chaincode is the central part of a distributed application in Fabric and may be written by an untrusted developer. Special chaincodes exist for managing the blockchain system and maintaining parameters, collectively called *system chaincodes* (Sec. 4.6).
- An *endorsement policy* that is evaluated in the *validation phase*. Endorsement policies cannot be chosen or modified by untrusted application developers; they are part of the system. An endorsement policy acts as a static library for transaction validation in Fabric, which can merely be parameterized by the chaincode. Only designated *administrators* may run system management functions and have the right to modify the endorsement policy.
  A typical endorsement policy lets the chaincode specify the endorsers for a transaction in the form of a set of peers that are necessary for endorsement; it uses a monotone logical expression on sets, such as "three out of five" or "A and B or B and C." Custom endorsement policies may implement arbitrary logic (e.g., our Bitcoin-inspired cryptocurrency in Sec. 5.1).

A client sends transactions to the peers specified by the endorsement policy. Each transaction is then executed by specific peers and its output is recorded; this step is also called *endorsement*. After execution, transactions enter the *ordering phase*, which uses a pluggable consensus protocol to produce a totally ordered sequence of endorsed transactions grouped in blocks. These are broadcast to all peers, with the (optional) help of gossip. Unlike standard active replication [31], which totally orders transaction *inputs*, Fabric orders transaction *outputs* combined with state dependencies, as computed during the execution phase. Each peer then validates the state changes from endorsed transactions with respect to the endorsement policy and the consistency of the execution in the *validation phase*. All peers validate the transactions in the same order and validation is deterministic. In this sense, Fabric introduces a novel *hybrid replication* paradigm in the Byzantine model, which combines passive replication (the pre-consensus computation of state updates) and active replication (the post-consensus validation of execution results and state changes).

The execute-order-validate approach is illustrated by Fig. 2.

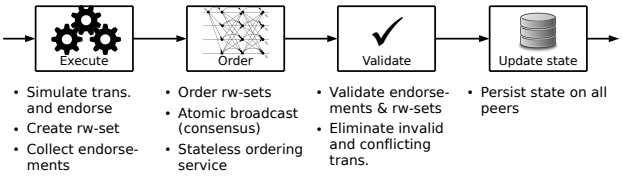

Figure 2. Execute-order-validate architecture of Fabric (*rw-set* means a readset and writeset as explained in Sec. 3.2).

A Fabric blockchain consists of a set of *nodes* that form a *network*. As Fabric is *permissioned*, all nodes that participate in the network have an identity, as provided by a modular *membership service provider (MSP)* (Sec. 4.1). Nodes in a Fabric network take up one of three roles:

Clients submit *transaction proposals* for *execution*, help orchestrate the execution phase, and, finally, broadcast *transactions* for ordering.

Peers execute transaction proposals and *validate* transactions. Peers also maintain the blockchain *ledger*, an append-only data structure recording all transactions in the form of a hash chain, as well as the *state*, a succinct representation of the latest ledger state. Not all peers execute all transaction proposals, only a subset of them called *endorsing peers* (or, simply, *endorsers*), as specified by the policy of the chaincode to which the transaction pertains. However, all peers maintain the complete ledger.

Orderering Service Nodes (OSN) (or, simply, *orderers*) are the nodes that collectively form the *ordering service*. In short, the ordering service establishes the *total order* of all transactions in Fabric, where each transaction contains state updates and dependencies computed during the execution phase, along with cryptographic sig-

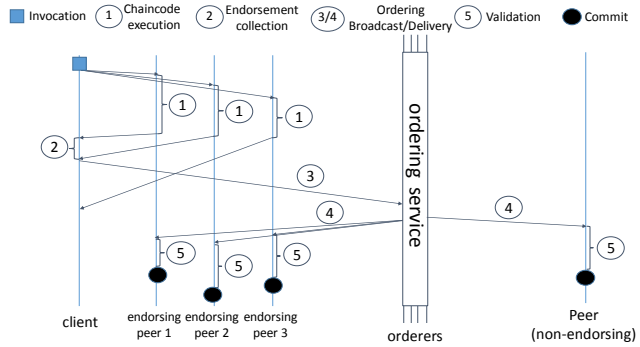

Figure 3. Fabric high level transaction flow.

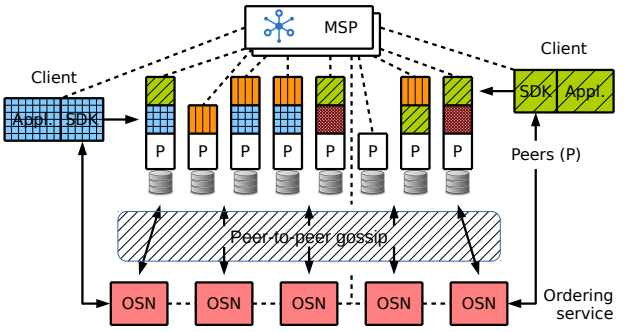

Figure 4. A Fabric network with federated MSPs and running multiple (differently shaded and colored) chaincodes, selectively installed on peers according to policy.

natures of the endorsing peers that computed them. Orderers are entirely unaware of the application state, and do not participate in the execution nor in the validation of transactions. This design choice renders consensus in Fabric as modular as possible and simplifies replacement of consensus protocols in Fabric.

Since it is possible to run one physical node with multiple roles, Fabric can also be operated like a traditional peer-to-peer blockchain system, in which every node maintains the state and invokes, validates, and orders transactions. The transaction flow in Fabric across the different types of nodes is depicted in Fig. 3.

Compared to the description focusing on one single blockchain so far, a Fabric network actually supports multiple blockchains connected to the same ordering service. Each such blockchain is called a *channel* and may have different peers as its members. Channels can be used to partition the state of the blockchain network, but consensus across channels is not coordinated and the total order of transactions in each channel is separate from the others. Certain deployments that consider all orderers as trusted may also implement by-channel access control for peers. In the following we mention channels only briefly and concentrate on one single channel.

The next three sections explain the transaction flow in Fabric and illustrate the steps of the execution, ordering, and validation phases. A Fabric network is shown in Fig. 4.

3.2 Execution Phase

In the execution phase, clients send the *transaction proposal* (or, simply, *proposal*) to one or more endorsers for execution. Recall that every chaincode implicitly specifies a set of endorsers via the endorsement policy. A proposal contains the identity of the submitting client (according to the MSP), the transaction payload in the form of an operation to execute, parameters, and the identifier of the chaincode to which it belongs, a nonce to be used only once by each client (such as a counter or a random value), and a transaction identifier derived from the client identifier and the nonce. The client also signs the proposal.

The endorsers *simulate* the proposal, by executing the operation on the specified chaincode, which has been installed on the blockchain. The chaincode runs in a Docker container, isolated from the main endorser process.

A proposal is simulated against the endorser's local blockchain state, without any synchronization with other peers at this point; moreover, endorsers do not persist the results of the simulation to the ledger state. The state of the blockchain is maintained by the *peer transaction manager (PTM)* in the form of a versioned key-value store, in which successive updates to a key have monotonically increasing version numbers (Sec. 4.4). The state created by a chaincode is scoped exclusively to that chaincode and cannot be accessed directly by another chaincode. Note that the chaincode is not supposed to maintain the local state in the program code, only what it maintains in the blockchain state that is accessed with *GetState*, *PutState*, and *DelState* operations. Given the appropriate permission, a chaincode may invoke another chaincode to access its state within the same channel.

As a result of the simulation, each endorser produces a value *writeset*, consisting of the state updates produced by simulation (i.e., the modified keys along with their new values), as well as a *readset*, representing the version dependencies of the proposal simulation (i.e., all keys read during simulation along with their version numbers). After the simulation, the endorser cryptographically signs a message called *endorsement*, which contains *readset* and *writeset* (together with metadata such as transaction ID, endorser ID, and endorser signature) and sends it back to the client in a *proposal response*. The client collects endorsements until they satisfy the endorsement policy of the chaincode, which the transaction invokes (see Sec. 3.4). In particular, this requires all endorsers as determined by the policy to produce the same execution result (i.e., identical *readset* and *writeset*). Then, the client proceeds to create the transaction and passes it to the ordering service.

Discussion on design choices. As the endorsers simulate the proposal without synchronizing with other endorsers, two endorsers may execute it on different states of the ledger and produce different outputs. For the standard endorsement policy which requires multiple endorsers to produce the same result, this implies that under high contention of operations accessing the same keys, a client may not be able to satisfy the endorsement policy. This is a new element compared to primary-backup replication in replicated databases with synchronization through middleware [24]: a consequence of the assumption that no single peer is trusted for correct execution in a blockchain.

We consciously adopted this design, as it considerably simplifies the architecture and is adequate for typical blockchain applications. As demonstrated by the approach of Bitcoin, distributed applications can be formulated such that contention by operations accessing the same state can be reduced, or eliminated completely in the normal case (e.g., in Bitcoin, two operations that modify the same "object" are not allowed and represent a double-spending attack [28]).

Executing a transaction before the ordering phase is critical to tolerating non-deterministic chaincodes as discussed in Section 2. A chaincode in Fabric with non-deterministic transactions can only endanger the liveness of its own operations, because a client cannot gather a sufficient number of endorsements, for instance. This is much more acceptable in practice than the situation in an order-execute architecture, where non-deterministic operations lead to inconsistencies in the state of the peers.

Finally, tolerating non-deterministic execution also addresses DoS attacks from untrusted chaincode as an endorser can simply abort an execution according to a local policy if it suspects a DoS attack. This will not endanger the consistency of the system, and again, such unilateral abortion of execution is not possible in order-execute architectures.

3.3 Ordering Phase

When a client has collected enough endorsements on a proposal, it assembles a *transaction* and submits this to the ordering service. The transaction contains the transaction payload (i.e., the chaincode operation including parameters), transaction metadata, and a set of endorsements. The ordering phase establishes a total order on all submitted transactions per channel. In other words, ordering atomically broadcasts [7] endorsements and thereby establishes consensus on transactions, despite faulty orderers. Moreover, the ordering service batches multiple transactions into *blocks* and outputs a hash-chained sequence of blocks containing transactions. Grouping or batching transactions into blocks improves the throughput of the broadcast protocol, which is a well-known technique in the context of fault-tolerant broadcasts.

At a high level, the interface of the ordering service only supports the following two operations. These operations are invoked by a peer and implicitly parameterized by a channel identifier:

- *broadcast*(tx): A client calls this operation to *broadcast* an arbitrary transaction tx, which usually contains the transaction payload and a signature of the client, for dissemination.
- $B \leftarrow$ *deliver*(s): A client calls this to retrieve block B with non-negative sequence number s. The block contains a list of transactions $[tx_1, \ldots, tx_k]$ and a hash-chain value h representing the block with sequence number $s-1$, i.e., $B = ([tx_1, \ldots, tx_k], h)$. As the client may call this multiple times and always returns the same block once it is available, we say the peer *delivers* block B with sequence number s when it receives B *for the first* time upon invoking *deliver*(s).

The ordering service ensures that the *delivered* blocks on one channel are totally ordered. More specifically, ordering ensures the following safety properties for each channel:

Agreement: For any two blocks B delivered with sequence number s and B' delivered with s' at correct peers such that $s = s'$, it holds $B = B'$.

Hashchain integrity: If some correct peer delivers a block B with number s and another correct peer delivers block $B' = ([tx_1, \ldots, tx_k], h')$ with number $s+1$, then it holds $h' = H(B)$, where $H(\cdot)$ denotes the cryptographic hash function.

No skipping: If a correct peer p delivers a block with number $s > 0$ then for each $i = 0, \ldots, s-1$, peer p has already delivered a block with number i.

No creation: When a correct peer delivers block B with number s, then for every $tx \in B$ some client has already broadcast tx.

For liveness, the ordering service supports at least the following "eventual" property:

Validity: If a correct client invokes *broadcast*(tx), then every correct peer eventually delivers a block B that includes tx, with some sequence number.

However, every individual ordering implementation is allowed to come with its own liveness and fairness guarantees with respect to client requests.

Since there may be a large number of peers in the blockchain network, but only relatively few nodes are expected to implement the ordering service, Fabric can be configured to use a built-in *gossip service* for disseminating delivered blocks from the ordering service to all peers (Sec. 4.3). The implementation of gossip is scalable and agnostic to the particular implementation of the ordering service, hence it works with both CFT and BFT ordering services, ensuring the modularity of Fabric.

The ordering service may also perform access control checks to see if a client is allowed to broadcast messages or receive blocks on a given channel. This and other features of the ordering service are further explained in Section 4.2.

Discussion on design choices. It is very important that the ordering service does not maintain any state of the blockchain, and neither validates nor executes transactions. This architecture is a crucial, defining feature of Fabric, and makes Fabric the first blockchain system to totally separate consensus from execution and validation. This makes consensus as modular as possible, and enables an ecosystem of consensus protocols implementing the ordering service.

3.4 Validation Phase

Blocks are delivered to peers either via a direct connection to the ordering service or through gossip. When a new block arrives, it enters the validation phase, consisting of three sequential steps:

1. The *endorsement policy evaluation* occurs in parallel for all transactions within the block. The evaluation is the task of the so-called *validation system chaincode (VSCC)*, a static library that is part of the blockchain's configuration and is responsible for validating the endorsement with respect to the endorsement policy configured for the chaincode (see Sec. 4.6). If the endorsement is not satisfied, the transaction is marked as invalid and its effects are disregarded.
2. A *read-write conflict check* is done for all transactions in the block sequentially. For each transaction it compares the versions of the keys in the *readset* field to those in the current state of the ledger, as stored locally by the peer, and ensures they are still the same. If the versions do not match, the transaction is marked as invalid and its effects are disregarded.
3. The *ledger update phase* runs last, in which the block is appended to the locally stored ledger and the blockchain state is updated. In particular, when adding the block to the ledger, the results of the validity checks in the first two steps are persisted as well, in the form of a bit mask denoting the transactions that are valid within the block. This facilitates the reconstruction of the state at a later time. Furthermore, all state updates are applied by writing all key-value pairs in *writeset* to the local state.

The default VSCC in Fabric allows monotone logical expressions over the set of endorsers configured for a chaincode to be expressed. The VSCC evaluation verifies that the set of peers, as expressed through valid signatures on endorsements of the transaction, satisfy the expression. Different VSCC policies can be configured statically, however.

Discussion on design choices. The ledger of Fabric contains all transactions, including those that are deemed invalid. This follows from the overall design, because ordering service, which is agnostic to chaincode state, produces the chain of the blocks and because the validation is done by the peers post-consensus. This feature is needed in certain use cases that require tracking of invalid transactions during subsequent audits, and stands in contrast to other blockchains (e.g., Bitcoin and Ethereum), where the ledger contains only valid transactions.

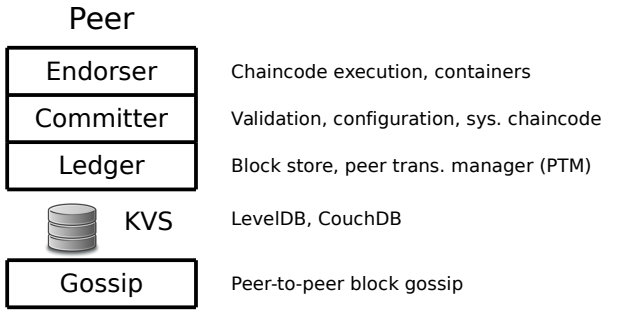

Figure 5. Components of a Fabric peer.

4. Fabric Components

Fabric is written in Go and uses the gRPC framework (`http://grpc.io/`) for communication between clients, peers, and orderers. In the following we describe some important components in more detail. Figure 5 shows the components of a peer.

4.1 Membership Service

The *membership service provider (MSP)* maintains the identities of all nodes in the system (clients, peers, and orderers) and is responsible for issuing node credentials that are used for authentication and authorization. Since Fabric is *permissioned*, all interactions among nodes occur through messages that are authenticated, typically with digital signatures. The membership service comprises a component at each node, where it may authenticate transactions, verify the integrity of transactions, sign endorsements, validate endorsements, and authenticate other blockchain operations. Tools for key management and registration of nodes are also part of the MSP.

The MSP is an abstraction for which different instantiations are possible. The default MSP implementation in Fabric handles standard PKI methods for authentication based on digital signatures and can accommodate commercial certification authorities (CAs). A stand-alone CA is provided as well with Fabric, called *Fabric-CA*. Furthermore, alternative MSP implementations are envisaged, such as one relying on anonymous credentials for authorizing a client to invoke a transaction without linking this to an identity [10].

Fabric allows two modes for setting up a blockchain network. In *offline mode*, credentials are generated by a CA and distributed out-of-band to all nodes. Peers and orderers can only be registered in offline mode. For enrolling clients, Fabric-CA provides an *online mode* that issues cryptographic credentials to them. The MSP configuration must ensure that all nodes, especially all peers, recognize the same identities and authentications as valid.

The MSP permits identity federation, for example, when multiple organizations operate a blockchain network. Each organization issues identities to its own members and every peer recognizes members of all organizations. This can be achieved with multiple MSP instantiations, for example, by creating a mapping between each organization and an MSP.

4.2 Ordering Service

The ordering service manages multiple channels. On every channel, it provides the following services:

1. *Atomic broadcast* for establishing order on transactions, implementing the *broadcast* and *deliver* calls (Sec. 3.3).
2. *Reconfiguration* of a channel, when its members modify the channel by broadcasting a *configuration update transaction* (Sec. 4.6).
3. Optionally, *access control*, in those configurations where the ordering service acts as a trusted entity, restricting broadcasting of transactions and receiving of blocks to specified clients and peers.

The ordering service is bootstrapped with a *genesis block* on the *system channel*. This block carries a *configuration transaction* that defines the properties of the ordering service.

The current production implementation consists of *ordering-service nodes (OSNs)* that implement the operations described here and communicate through the system channel. The actual atomic broadcast function is provided by an instance of Apache Kafka (`http://kafka.apache.org`), which offers scalable publish-subscribe messaging and strong consistency despite node crashes, based on ZooKeeper. Kafka may run on physical nodes separate from the OSNs. The OSNs act as proxies between the peers and Kafka.

An OSN directly injects a newly received transaction to the atomic broadcast (e.g., to the Kafka broker). On the other hand, the nodes *batch* transactions received from the atomic broadcast and *form blocks*. A block is *cut* as soon as one of three conditions is met: (1) the block contains the specified maximal number of transactions; (2) the block has reached a maximal size (in bytes); or (3) an amount of time has elapsed since the first transaction of a new block was received, as explained below.

This batching process is *deterministic* and therefore produces the same blocks at all nodes. It is easy to see that the first two conditions are trivially deterministic, given the stream of transactions received from the atomic broadcast. To ensure deterministic block production in the third case, a node starts a timer when it reads the first transaction in a block from the atomic broadcast. If the block is not yet cut when the timer expires, the node broadcasts a special *time-to-cut* transaction on the channel, which indicates the sequence number of the block which it intends to cut. On the other hand, every node immediately cuts a new block upon receiving the first *time-to-cut* transaction for the given block number. Since this transaction is atomically delivered to all connected nodes, they all include the same list of transactions in the block. (To deploy this scheme in the presence of f Byzantine-faulty OSNs, the block is cut only when receiving $f + 1$ *time-to-cut* transactions.)

The orderers persist a range of the most recently delivered blocks directly to their filesystem, so they can answer to peers retrieving blocks through *deliver*.

The orderer using Kafka is one of three ordering service implementations currently available. A centralized orderer, called *Solo*, runs on one node and is used for development. A proof-of-concept orderer based on *BFT-SMaRt* [3] has also been made available [34]; it ensures the atomic broadcast service, but not yet reconfiguration and access control. This illustrates the modularity of consensus in Fabric.

4.3 Peer Gossip

One advantage of separating the execution, ordering, and validation phases is that they can be scaled independently. However, since most consensus algorithms (in the CFT and BFT models) are bandwidth-bound, the throughput of the ordering service is capped by the network capacity of its nodes. Consensus cannot be scaled up by adding more nodes [14, 36], rather, throughput will decrease. However, since ordering and validation are decoupled, we are interested in efficiently broadcasting the execution results to all peers for validation, after the ordering phase. This is precisely the goal of the gossip component, which utilizes epidemic multicast [15] for this purpose. The blocks are signed by the ordering service. This means that a peer can, upon receiving all blocks, independently assemble the blockchain and verify its integrity.

Dissemination through gossip is robust and resistant to the node failures, in contrast to overlay networks. Making random selections allows gossip to reduce the overhead of maintaining connectivity among peers, and, moreover, reduces the attack surface. Gossip works well in the permissioned environment, where it can withstand sybil attacks and message forgery.

The communication layer for gossip is based on gRPC and utilizes TLS with mutual authentication, which enables each side to bind the TLS credentials to the identity of the remote peer. The gossip component maintains an up-to-date *membership view* of the online peers in the system. All peers independently build a local view from periodically disseminated membership data. Furthermore, a peer can reconnect to the view after a crash or a network outage.

The main purpose of gossip is to reliably distribute messages, i.e., blocks from the ordering phase, among the peers, taking advantage of a *push-pull* protocol. It uses two phases for information dissemination: during *push*, each peer selects a random set of active neighbors from the membership view, and forwards them the message; during *pull*, each peer periodically probes a set of randomly selected peers and re-

quests missing messages. It has been shown [15, 22] that using both methods in tandem is crucial to optimally utilize the available bandwidth and to ensure that all peers receive all messages with high probability.

In order to reduce the load of sending blocks from the ordering nodes to the network, the protocol also *elects a leader peer* that pulls blocks from the ordering service on their behalf and initiates the gossip distribution. This mechanism is resilient to leader failures.

Another task of gossip is to *transfer the state* to newly joining peers and peers that were disconnected for a long time. They need to receive all blocks in the chain. This feature relies on the fact that the largest block-sequence number stored by each peer is disseminated with the membership data.

4.4 Ledger

The ledger component at each peer maintains the ledger and the blockchain state on persistent storage and enables *simulation*, *validation*, and *ledger-update* phases. Broadly, it consists of a *block store* and a *peer transaction manager (PTM)*.

Ledger block store. The ledger block store persists transaction blocks and is implemented as a set of append-only files. Since the blocks are immutable and arrive in a definite order, an append-only structure gives maximum performance. In addition, the block store maintains a few indices for random access to a block or to a transaction in a block.

Peer transaction manager (PTM). The PTM maintains the *latest state* in a versioned key-value store. It stores one tuple of the form (key, val, ver) for each unique entry *key* stored by any chaincode, containing its most recently stored value *val* and its latest version *ver*. The version consists of the block sequence number and the sequence number of the transaction (that stores the entry) within the block. This makes the version unique and monotonically increasing.

The PTM uses a local key-value store to realize the versioned key-value store, implemented by a LevelDB key-value database implemented in Go (`https://github.com/syndtr/goleveldb`) or Apache CouchDB (`http://couchdb.apache.org/`).

During simulation the PTM provides a stable snapshot of the latest state to the transaction. As mentioned in Section 3.2, the PTM records in *readset* a tuple (key, ver) for each entry accessed by *GetState* and in *writeset* a tuple (key, val) for each entry updated with *PutState* by the transaction. In addition, the PTM supports range queries, for which it computes a cryptographic hash of the query results (a set of tuples (key, ver)) and adds the query string itself and the hash to *readset*.

For transaction validation (Sec. 3.4), the PTM validates all transactions in a block sequentially. This checks whether a transaction conflicts with any preceding transaction (within the block or earlier). For any key in *readset*, if the version recorded in *readset* differs from the version present in the latest state (assuming that all preceding valid transactions are committed), then the PTM marks the transaction as invalid. For range queries, the PTM re-executes the query and compares the hash with the one present in *readset*, to ensure that no phantom reads occur. This read-write conflict semantics results in one-copy serializability [23].

The ledger component tolerates a crash of the peer during the ledger update as follows. After receiving a new block, the PTM has already performed validation and marked transactions as valid or invalid within the block, using a bit mask as mentioned in Section 3.4. The ledger now writes the block to the ledger block store, flushes it to disk, and subsequently updates the block store indices. Then the PTM applies the state changes from *writeset* of all valid transactions to the local versioned store. Finally, it computes and persists a value *savepoint*, which denotes the largest successfully applied block number. The value *savepoint* is used to recover the indices and the latest state from the persisted blocks when recovering from a crash.

4.5 Chaincode Execution

Chaincode is executed within an environment that is loosely coupled with the rest of the peer and that supports plugins for adding new languages for programming chaincodes. Currently three languages are supported for chaincode: Go, Java, and Node.

Every user-level or application chaincode runs in a separate process within a Docker container environment, which isolates the chaincodes from each other and from the peer. This also simplifies the management of the lifecycle for chaincodes (i.e., starting, stopping, or aborting chaincode). The chaincode and the peer communicate using gRPC messages. Through this loose coupling, the peer is agnostic of the actual language in which chaincode is implemented.

In contrast to application chaincode, system chaincode runs directly in the peer process. System chaincode can implement specific functions needed by Fabric and may be used in situations where the isolation among user chaincodes is overly restrictive. More details on system chaincodes are given in the next section.

4.6 Configuration and System Chaincodes

Fabric's basic behavior is customized through *channel configuration* and through special chaincodes, known as *system chaincodes*.

Channel configuration. Recall that a channel forms one logical blockchain. The configuration of a channel is maintained in metadata persisted in special *configuration blocks*. Each configuration block contains the full channel configuration and does not contain any other transactions. Each blockchain begins with a configuration block known as the *genesis block* which is used to bootstrap the channel. The channel configuration includes:

- Definitions of the MSPs for the participating nodes.
- The network addresses of the OSNs.
- Shared configuration for the consensus implementation and the ordering service, such as batch size and timeouts.
- Rules governing access to the ordering service operations (*broadcast*, and *deliver*).
- Rules governing how each part the channel configuration may be modified.

The configuration of a channel may be updated using a *channel configuration update transaction*. This transaction contains a representation of the changes to be made to the configuration, as well as a set of signatures. The ordering service nodes evaluate whether the update is valid by using the current configuration to verify that the modifications are authorized using the signatures. The orderers then generate a new configuration block, which embeds the new configuration and the configuration update transaction. Peers receiving this block validate whether the configuration update is authorized based on the current configuration; if valid, they update their current configuration.

System chaincodes. The application chaincodes are deployed with a reference to an *endorsement system chaincode (ESCC)* and to a *validation system chaincode (VSCC)*. These two chaincodes are selected in a symmetric way, such that the output of the ESCC (an endorsement) may be validated as part of the input to the VSCC.

The ESCC takes as input a proposal and the proposal simulation results. If the results are satisfactory, then the ESCC produces a response, containing the results and the endorsement. For the default ESCC, this endorsement is simply a signature by the peer's local signing identity.

The VSCC takes as input a transaction and outputs whether that transaction is valid. For the default VSCC, the endorsements are collected and evaluated against the endorsement policy specified for the chaincode.

Further system chaincodes implement other support functions, such as configuration and chaincode lifecycle.

5. Evaluation

Even though Fabric is not yet performance-tuned and optimized, we report in this section on some preliminary performance numbers. Fabric is a complex distributed system; its performance depends on many parameters including the choice of a distributed application and transaction size, the ordering service and consensus implementation and their parameters, the network parameters and topology of nodes in the network, the hardware on which nodes run, the number of nodes and channels, further configuration parameters, and the network dynamics. Therefore, in-depth performance evaluation of Fabric is postponed to future work.

In the absence of a standard benchmark for blockchains, we use the most prominent blockchain application for evaluating Fabric, a simple authority-minted cryptocurrency that uses the data model of Bitcoin, which we call *Fabric coin* (abbreviated hereafter as *Fabcoin*). This allows us to put the performance of Fabric in the context of other permissioned blockchains, which are often derived from Bitcoin or Ethereum. For example, it is also the application used in benchmarks of other permissioned blockchains [19, 32].

In the following, we first describe Fabcoin (Sec. 5.1), which also demonstrates how to customize the validation phase and endorsement policy. In Section 5.2 we present the benchmark and discuss our results.

5.1 Fabric Coin (Fabcoin)

UTXO cryptocurrencies. The data model introduced by Bitcoin [28] has become known as "unspent transaction output" or *UTXO*, and is also used by many other cryptocurrencies and distributed applications. UTXO represents each step in the evolution of a data object as a separate atomic state on the ledger. Such a state is created by a transaction and destroyed (or "consumed") by another unique transaction occurring later. Every given transaction destroys a number of *input states* and creates one or more *output states*. A "coin" in Bitcoin is initially created by a *coinbase* transaction that rewards the "miner" of the block. This appears on the ledger as a *coin state* designating the miner as the owner. Any coin can be *spent* in the sense that the coin is assigned to a new owner by a transaction that atomically destroys the current coin state designating the previous owner and creates another coin state representing the new owner.

We capture the UTXO model in the key-value store of Fabric as follows. Each UTXO state corresponds to a unique KVS entry that is created once (the coin state is "unspent") and destroyed once (the coin state is "spent"). Equivalently, every state may be seen as a KVS entry with logical version 0 after creation; when it is destroyed again, it receives version 1. There should not be any concurrent updates to such entries (e.g., attempting to update a coin state in different ways amounts to double-spending the coin).

Value in the UTXO model is transferred through transactions that refer to several input states that all belong to the entity issuing the transaction. An entity owns a state because the public key of the entity is contained in the state itself. Every transaction creates one or more output states in the KVS representing the new owners, deletes the input states in the KVS, and ensures that the sum of the values in the input states equals the sum of the output states' values. There is also a policy determining how value is created (e.g., *coinbase* transactions in Bitcoin or specific *mint* operations in other systems) or destroyed (i.e., as a fee consumed by the execution).

Fabcoin implementation. Each state in Fabcoin is a tuple of the form $(key, val) = (txid.j, (amount, owner, label))$, denoting the coin state created as the j-th output of a transaction with identifier *txid* and allocating *amount* units labeled with *label* to the entity whose public key is *owner*. Labels are

strings used to identify a given type of a coin (e.g., 'USD', 'EUR', 'FBC'). Transaction identifiers are short values that uniquely identify every Fabric transaction. The Fabcoin implementation consists of three parts: (1) a client wallet, (2) the Fabcoin chaincode, and (3) a custom VSCC for Fabcoin implementing its endorsement policy.

Client wallet. By default, each Fabric client maintains a *Fabcoin wallet* that locally stores a set of cryptographic keys allowing the client to spend coins. For creating a SPEND transaction that transfers one or more coins, the client wallet creates a Fabcoin request $request = (inputs, outputs, sigs)$ containing: (1) a list of *input coin states*, $inputs = [in, \ldots]$ that specify coin states $(in, \ldots)$ the client wishes to spend, as well as (2) a list of *output coin states*, $outputs = [(amount, owner, label), \ldots]$. The client wallet signs, with the private keys that correspond to the input coin states, the concatenation of the Fabcoin request and a nonce, which is a part of every Fabric transaction, and adds the signatures in a set *sigs*. A SPEND transaction is valid when the sum of the amounts in the input coin states is at least the sum of the amounts in the outputs and when the output amounts are positive. For a MINT transaction that creates new coins, *inputs* contains only an identifier (i.e., a reference to a public key) of a special entity called *Central Bank (CB)*, whereas *outputs* contains an arbitrary number of coin states. To be considered valid, the signatures of a MINT transaction in *sigs* must be a cryptographic signature under the public key of CB over the concatenation of the Fabcoin request and of the aforementioned nonce. Fabcoin may be configured to use multiple CBs or specify a threshold number of signatures from a set of CBs. Finally, the client wallet includes the Fabcoin request into a transaction and sends this to a peer of its choice.

Fabcoin chaincode. A peer runs the chaincode of Fabcoin which simulates the transaction and creates readsets and writesets. In a nutshell, in the case of a SPEND transaction, for every input coin state $in \in inputs$ the chaincode first performs $GetState(in)$; this puts *in* into the readset along with its current version in Fabric's versioned KVS (Sec. 4.4). Then the chaincode executes $DelState(in)$ for every input state *in*, which also adds *in* to the writeset and effectively marks the coin state as "spent." Finally, for $j = 1, \ldots, |outputs|$, the chaincode executes $PutState(txid.j, out)$ with the j-th output $out = (amount, owner, label)$ in *outputs*. In addition, a peer optionally runs the transaction validation code as described next in the VSCC step for Fabcoin; this is not necessary, since the custom VSCC actually validates transactions, but it allows the (correct) peers to filter out potentially malformed transactions. In our implementation, the chaincode runs the Fabcoin VSCC without cryptographically verifying the signatures.

Custom VSCC. Finally, every peer validates Fabcoin transactions using custom VSCC. This verifies first the cryptographic signature(s) in *sigs* under the respective public key(s) and performs semantic validation as follows. For a MINT transaction, it checks that the output states are created under the matching transaction identifier (*txid*) and that all output amounts are positive. For a SPEND transaction, the VSCC additionally verifies (1) that for all input coin states, an entry in the readset has been created and that it was also added to the writeset and marked as deleted, (2) that the sum of the amounts for all input coin states equals the sum of amounts of all output coin states, and (3) that input and output coin labels match. Here, the VSCC obtains the amounts for the input coins by retrieving their current values from the ledger.

Notice that the Fabcoin VSCC does not check transactions for double spending, as this occurs through Fabric's standard validation that runs after the custom VSCC. In particular, if two transactions attempt to assign the same unspent coin state to a new owner, both would pass the VSCC logic but would be caught subsequently in the read-write conflict check performed by the PTM. According to Sections 3.4 and 4.4, the PTM verifies that the current version number stored in the ledger matches the one in the readset; hence, after the first transaction has changed the version of the coin state, the transaction ordered second will be recognized as invalid.

5.2 Experiments

Setup. Unless explicitly mentioned differently, in our experiments: (1) nodes run on Fabric version v1.1-preview[2] instrumented for performance evaluation through local logging, (2) nodes are hosted in a single IBM Cloud (SoftLayer) datacenter as dedicated VMs interconnected with 1Gbps networking, (3) all nodes are 2.0 GHz 16-vCPU VMs running Ubuntu with 8GB of RAM and SSDs as local disks, (4) a single-channel ordering service runs a typical Kafka orderer setup with 3 ZooKeeper nodes, 4 Kafka brokers and 3 Fabric orderers, all on distinct VMs, (5) there are 5 peers in total, all of which are Fabcoin endorsers, and (6) signatures use the default 256-bit ECDSA scheme. In order to measure and stage latencies in the transaction flow spanning multiple nodes, the node clocks are synchronized with an NTP service throughout the experiments. All communication among Fabric nodes is configured to use TLS.

Methodology. In every experiment, in the first phase we invoke transactions that contain only Fabcoin MINT operations to produce the coins, and then run a second phase of the experiment in which we invoke Fabcoin SPEND operation on previously minted coins (effectively running single-input, single-output SPEND transactions). When reporting throughput measurements, we use an increasing number of Fabric CLI clients (modified to issue concurrent requests) running on a single VM, until the end-to-end throughput is saturated, and state the throughput just before saturation. Throughput numbers are reported as an average throughput

[2] Patched with commit ID $9e770062$ in the Fabric *master* branch.

measured throughout the duration of an experiment, disregarding the "tail" of the experiment, where throughput drops due to some client threads finishing submitting their share of transactions. In every experiment, client threads submit at least 500k MINT and SPEND transactions.

Choosing the block size. A critical Fabric configuration parameter that impacts both throughput and latency is block size. To fix the block size for subsequent experiments, and to evaluate the impact of block size on performance, we ran experiments varying block size from 0.5MB to 4MB. Results are depicted in Fig. 6, showing peak throughput measured at the peers along with the corresponding average end-to-end (e2e) latency.

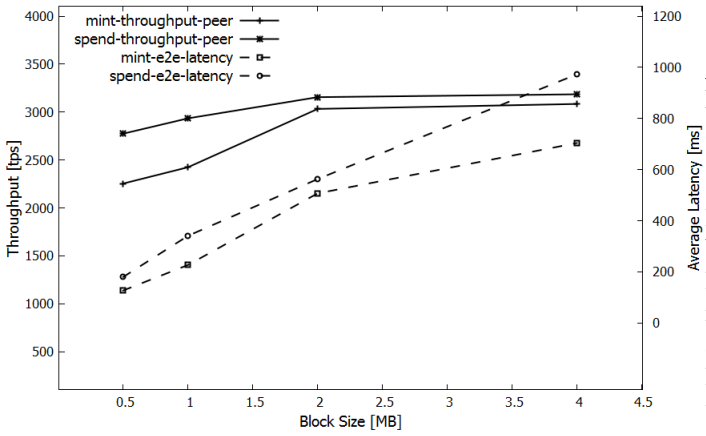

Figure 6. Impact of block size on throughput and latency.

We can observe that throughput does not significantly improve beyond a block size of 2MB, but latency gets worse (as expected). Therefore, we adopt 2MB as the block size for the following experiments, with the goal of maximizing the measured throughput, assuming the end-to-end latency of roughly 500ms is acceptable.

Size of transactions. During this experiment, we also observed the size MINT and SPEND transactions. In particular, the 2MB-blocks contained 473 MINT or 670 SPEND transactions, i.e., the average transaction size is 3.06kB for SPEND and 4.33kB for MINT. In general, transactions in Fabric are large because they carry certificate information. Besides, MINT transactions of Fabcoin are larger than SPEND transactions because they carry CB certificates. This is an avenue for future improvement of both Fabric and Fabcoin.

Impact of peer CPU. Fabric peers run many CPU-intensive cryptographic operations. To estimate the impact of CPU on throughput, we performed a set of experiments in which 4 peers run on 4, 8, 16 and 32 vCPU VMs, while also doing coarse-grained latency staging of block validation to better identify bottlenecks. Our experiment focused on the validation phase, as ordering with Kafka ordering service has never been a bottleneck in our experiments. The validation phase, and in particular the VSCC validation of Fabcoin, is computationally intensive, due to many digital signature verifications performed in this phase.

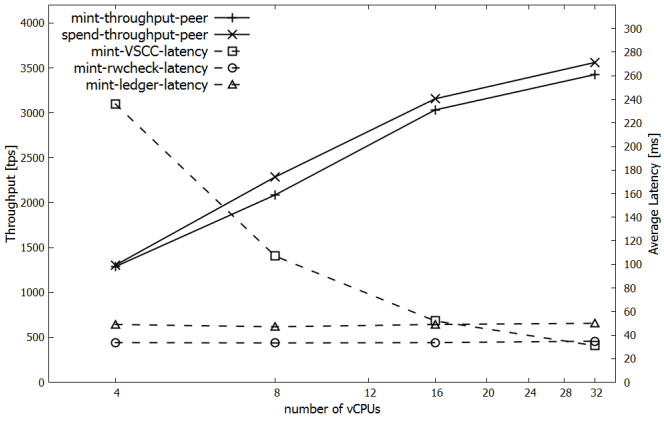

Figure 7. Impact of peer CPU on end-to-end throughput and block validation latency.

The results, with 2MB blocks, are shown in Fig. 7. SPEND latency is not shown in Fig. 7, as it scales similarly to MINT. We can observe that Fabcoin VSCC validation scales quasi-linearly with CPU, as Fabric VSCC validation is embarrassingly parallel. However, the read-write-check and ledger-access stages are sequential and become dominant with a larger number of cores (vCPUs). This suggests that future versions of Fabric could profit from pipelining validation stages (which are now sequential), optimizing stable-storage access, and parallelizing read-write dependency checks.

Finally, in this experiment, we measured over 3560 transactions per second (tps) average SPEND throughput at the 32-vCPU peer. The MINT throughput is, in general, slightly lower than that of SPEND, but the difference remains within 10%.

Latency profiling by stages. We further performed coarse-grained profiling of latency during our previous experiment at the peak reported throughput. Results are depicted in Table 1. The ordering phase comprises broadcast-deliver latency and internal latency within a peer before validation starts. The table reports average latencies for MINT and SPEND, standard deviation, and tail latencies (99% and 99.9%).

We can observe that ordering dominates the overall latency. We observe that average latencies are below 550ms with sub-second tail latencies. In particular, the highest end-to-end latencies in our experiment come from the first blocks, during the load build-up. Latency under lower load can be regulated and reduced by leveraging the time-to-cut parameter of the orderer (see Sec. 3.3), which we basically do not use in our experiments, as we set it to a large value.

SSD vs. RAM disk. To evaluate the overhead of using local SSDs for stable storage, we repeated the previous experiment with RAM disks (tmpfs) mounted as stable storage at

all peer VMs. The benefits are limited, as tmpfs only helps with the ledger phase of the validation at the peer. We measured sustained peak throughput at 3870 SPEND tps at 32-vCPU peer, roughly a 9% improvement over SSD.

	avg	st.dev	99%	99.9%
(1) endorsement	5.6 / 7.5	2.4 / 4.2	15 / 21	19 / 26
(2) ordering	248 / 365	60.0 / 92.0	484 / 624	523 / 636
(3) VSCC val.	31.0 / 35.3	10.2 / 9.0	72.7 / 57.0	113 / 108.4
(4) R/W check	34.8 / 61.5	3.9 / 9.3	47.0 / 88.5	59.0 / 93.3
(5) ledger	50.6 / 72.2	6.2 / 8.8	70.1 / 97.5	72.5 / 105
(6) validation (3+4+5)	116 / 169	12.8 / 17.8	156 / 216	199 / 230
(7) end-to-end (1+2+6)	371 / 542	63 / 94	612 / 805	646 / 813

Table 1. Latency statistics in milliseconds (ms) for MINT and SPEND, broken down into five stages at a 32-vCPU peer with 2MB blocks. Validation (6) comprises stages 3, 4, and 5; the end-to-end latency contains stages 1–5.

6. Related Work

The architecture of Fabric resembles that of a middleware-replicated database as pioneered by Kemme and Alonso [24]. However, all existing work on this addressed only crash failures, not the setting of distributed trust that corresponds to a BFT system. For instance, a replicated database with asymmetric update processing [25, Sec. 6.3] relies on one node to execute each transaction, which would not work on a blockchain. The execute-order-validate architecture of Fabric can be interpreted as a generalization of this work to the Byzantine model, with practical applications to distributed ledgers.

Byzantium [17] and HRDB [35] are two further predecessors of Fabric from the viewpoint of BFT database replication. Byzantium allows transactions to run in parallel and uses active replication, but totally orders BEGIN and COMMIT/ROLLBACK using a BFT middleware. In its optimistic mode, every operation is coordinated by a single master replica; if the master is suspected to be Byzantine, all replicas execute the transaction operations for the master and it triggers a costly protocol to change the master. HRDB relies in an even stronger way on a correct master. In contrast to Fabric, both systems use active replication, cannot handle a flexible trust model, and rely on deterministic operations by the replicas. However, their database API is richer than the KVS model of Fabric.

In *Eve* [21] a related architecture for SMR has been explored also in the BFT model. Its peers execute transactions concurrently and then verify that they all reach the same output state, using a consensus protocol. If the states diverge, they roll back and execute operations sequentially. Eve contains the element of independent execution, which also exists in Fabric, but offers none of its other features.

A large number of distributed ledger platforms in the permissioned model have come out recently, which makes it impossible to compare to all (some prominent ones are Tendermint [5], Quorum [19], Chain Core [12], Multichain (`https://www.multichain.com/`), Hyperledger Sawtooth (`https://sawtooth.hyperledger.org`), the Volt proposal [32], and more, see references in recent overviews [9, 16]). All platforms follow the order-execute architecture, as discussed in Section 2. As a representative example, take the Quorum platform [19], an enterprise-focused version of Ethereum. With its consensus based on *Raft* [29], it disseminates a transaction to all peers using gossip and the Raft leader (called *minter*) assembles valid transactions to a block, and distributes this using Raft. All peers execute the transaction in the order decided by the leader. Therefore it suffers from the limitations mentioned in Sections 1–2.

References

[1] P.-L. Aublin, R. Guerraoui, N. Knežević, V. Quéma, and M. Vukolić. The next 700 BFT protocols. *ACM Trans. Comput. Syst.*, 32(4):12:1–12:45, Jan. 2015.

[2] E. Ben-Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *IEEE Symposium on Security & Privacy*, pages 459–474, 2014.

[3] A. N. Bessani, J. Sousa, and E. A. P. Alchieri. State machine replication for the masses with BFT-SMART. In *International Conference on Dependable Systems and Networks (DSN)*, pages 355–362, 2014.

[4] G. Bracha and S. Toueg. Asynchronous consensus and broadcast protocols. *J. ACM*, 32(4):824–840, 1985.

[5] E. Buchman. Tendermint: Byzantine fault tolerance in the age of blockchains. M.Sc. Thesis, University of Guelph, Canada, June 2016.

[6] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg. The primary-backup approach. In S. Mullender, editor, *Distributed Systems (2nd Ed.)*, pages 199–216. ACM Press/Addison-Wesley, 1993.

[7] C. Cachin, R. Guerraoui, and L. E. T. Rodrigues. *Introduction to Reliable and Secure Distributed Programming (2. ed.)*. Springer, 2011.

[8] C. Cachin, S. Schubert, and M. Vukolić. Non-determinism in byzantine fault-tolerant replication. In *20th International Conference on Principles of Distributed Systems (OPODIS)*, 2016.

[9] C. Cachin and M. Vukolić. Blockchain consensus protocols in the wild. In A. W. Richa, editor, *31st Intl. Symposium on Distributed Computing (DISC 2017)*, pages 1:1–1:16, 2017.

[10] J. Camenisch and E. V. Herreweghen. Design and implementation of the *idemix* anonymous credential system. In *ACM Conference on Computer and Communications Security (CCS)*, pages 21–30, 2002.

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

[12] Chain. Chain protocol whitepaper. `https://chain.com/docs/1.2/protocol/papers/whitepaper`, 2017.

[13] B. Charron-Bost, F. Pedone, and A. Schiper, editors. *Replication: Theory and Practice*, volume 5959 of *Lecture Notes in Computer Science*. Springer, 2010.

[14] K. Croman, C. Decker, I. Eyal, A. E. Gencer, A. Juels, A. Kosba, A. Miller, P. Saxena, E. Shi, E. G. Sirer, et al. On scaling decentralized blockchains. In *International Conference on Financial Cryptography and Data Security (FC)*, pages 106–125. Springer, 2016.

[15] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic algorithms for replicated database maintenance. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 1–12. ACM, 1987.

[16] T. T. A. Dinh, R. Liu, M. Zhang, G. Chen, B. C. Ooi, and J. Wang. Untangling blockchain: A data processing view of blockchain systems. e-print, arXiv:1708.05665 [cs.DB], 2017.

[17] R. Garcia, R. Rodrigues, and N. M. Preguiça. Efficient middleware for Byzantine fault tolerant database replication. In *European Conference on Computer Systems (EuroSys)*, pages 107–122, 2011.

[18] R. Guerraoui, R. R. Levy, B. Pochon, and V. Quéma. Throughput optimal total order broadcast for cluster environments. *ACM Trans. Comput. Syst.*, 28(2):5:1–5:32, 2010.

[19] J. P. Morgan. Quorum whitepaper. `https://github.com/jpmorganchase/quorum-docs`, 2016.

[20] F. P. Junqueira, B. C. Reed, and M. Serafini. Zab: High-performance broadcast for primary-backup systems. In *International Conference on Dependable Systems & Networks (DSN)*, pages 245–256, 2011.

[21] M. Kapritsos, Y. Wang, V. Quéma, A. Clement, L. Alvisi, and M. Dahlin. All about Eve: Execute-verify replication for multi-core servers. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 237–250, 2012.

[22] R. Karp, C. Schindelhauer, S. Shenker, and B. Vocking. Randomized rumor spreading. In *Symposium on Foundations of Computer Science (FOCS)*, pages 565–574. IEEE, 2000.

[23] B. Kemme. One-copy-serializability. In *Encyclopedia of Database Systems*, pages 1947–1948. Springer, 2009.

[24] B. Kemme and G. Alonso. A new approach to developing and implementing eager database replication protocols. *ACM Transactions on Database Systems*, 25(3):333–379, 2000.

[25] B. Kemme, R. Jiménez-Peris, and M. Patiño-Martínez. *Database Replication*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2010.

[26] A. E. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *37th IEEE Symposium on Security & Privacy*, 2016.

[27] S. Liu, P. Viotti, C. Cachin, V. Quéma, and M. Vukolić. XFT: practical fault tolerance beyond crashes. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 485–500, 2016.

[28] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. `http://www.bitcoin.org/bitcoin.pdf`, 2009.

[29] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *USENIX Annual Technical Conference (ATC)*, pages 305–320, 2014.

[30] F. Pedone and A. Schiper. Handling message semantics with Generic Broadcast protocols. *Distributed Computing*, 15(2):97–107, 2002.

[31] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, 1990.

[32] S. Setty, S. Basu, L. Zhou, M. L. Roberts, and R. Venkatesan. Enabling secure and resource-efficient blockchain networks with VOLT. Technical Report MSR-TR-2017-38, Microsoft Research, 2017.

[33] A. Singh, T. Das, P. Maniatis, P. Druschel, and T. Roscoe. BFT protocols under fire. In *Symposium on Networked Systems Design & Implementation (NSDI)*, pages 189–204, 2008.

[34] J. Sousa, A. Bessani, and M. Vukolić. A Byzantine fault-tolerant ordering service for the Hyperledger Fabric blockchain platform. Technical Report arXiv:1709.06921, CoRR, 2017.

[35] B. Vandiver, H. Balakrishnan, B. Liskov, and S. Madden. Tolerating Byzantine faults in transaction processing systems using commit barrier scheduling. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 59–72, 2007.

[36] M. Vukolić. The quest for scalable blockchain fabric: Proof-of-work vs. BFT replication. In *International Workshop on Open Problems in Network Security (iNetSec)*, pages 112–125, 2015.

[37] J. Yin, J. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. Separating agreement from execution for Byzantine fault tolerant services. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 253–267, 2003.