Blockchain Consensus Protocols in the Wild[*]

Christian Cachin[1] and Marko Vukolić[2]

1 **IBM Research - Zürich, Rüschlikon, Switzerland**
`cca@zurich.ibm.com`
2 **IBM Research - Zürich, Rüschlikon, Switzerland**
`mvu@zurich.ibm.com`

Abstract

A blockchain is a distributed ledger for recording transactions, maintained by many nodes without central authority through a distributed cryptographic protocol. All nodes validate the information to be appended to the blockchain, and a consensus protocol ensures that the nodes agree on a unique order in which entries are appended. Consensus protocols for tolerating Byzantine faults have received renewed attention because they also address blockchain systems. This work discusses the process of assessing and gaining confidence in the resilience of a consensus protocols exposed to faults and adversarial nodes. We advocate to follow the established practice in cryptography and computer security, relying on public reviews, detailed models, and formal proofs; the designers of several practical systems appear to be unaware of this. Moreover, we review the consensus protocols in some prominent permissioned blockchain platforms with respect to their fault models and resilience against attacks.

1998 ACM Subject Classification C.2.4 Distributed Systems, D.1.3 Concurrent Programming

Keywords and phrases Permissioned blockchains, consensus, Byzantine fault-tolerance, snake oil, protocol analysis

Digital Object Identifier 10.4230/LIPIcs.DISC.2017.1

Category Keynote talk

1 Introduction

Blockchains or *distributed ledgers* are systems that provide a trustworthy service to a group of *nodes* or parties that do not fully trust each other. They stand in the tradition of distributed protocols for secure multiparty computation in cryptography and replicated services tolerating Byzantine faults in distributed systems. Blockchains also contain many elements from cryptocurrencies, although a blockchain system can be conceived without a currency or value tokens. Generally, the blockchain acts as a trusted and dependable third party, for maintaining shared state, mediating exchanges, and providing a secure computing engine. Many blockchains can execute arbitrary tasks, typically called *smart contracts*, written in a domain-specific or a general-purpose programming language.

In a *permissionless blockchain*, such as Bitcoin or Ethereum, anyone can be a user or run a node, anyone can "write" to the shared state through invoking transactions (provided transaction fees are paid for), and anyone can participate in the consensus process for

[*] This work has been supported in part by the European Commission through the Horizon 2020 Framework Programme (H2020-ICT-2014-1) under grant agreements number 643964-SUPERCLOUD and 644579 ESCUDO-CLOUD and in part by the Swiss State Secretariat for Education, Research and Innovation (SERI) under contracts number 15.0091 and 15.0087.

determining the “valid” state. A *permissioned blockchain* in contrast, is operated by known entities, such as in *consortium blockchains*, where members of a consortium or stakeholders in a given business context operate a permissioned blockchain network. Permissioned blockchains systems have means to identify the nodes that can control and update the shared state, and often also have ways to control who can issue transactions. A *private blockchain* is a special permissioned blockchain operated by one entity, i.e., within one single trust domain.

Permissioned blockchains address many of the problems that have been studied in the field of distributed computing over decades, most prominently for developing *Byzantine fault-tolerant (BFT)* systems. Such blockchains can benefit from many techniques developed for reaching consensus, replicating state, broadcasting transactions and more, in environments where network connectivity is uncertain, nodes may crash or become subverted by an adversary, and interactions among nodes are inherently asynchronous. The wide-spread interest in blockchain technologies has triggered new research on practical distributed consensus protocols. There is also a growing number of startups, programmers, and industry groups developing blockchain protocols based on their own ideas, not relying on established knowledge.

The purpose of this paper is to give an overview of consensus protocols actually being used in the context of permissioned blockchains, to review the underlying principles, and to compare the resilience and trustworthiness of some protocols. We leave out permissionless (or “public”) blockchains that are coupled to a cryptocurrency and their consensus protocols, such as *proof-of-work* or *proof-of-stake*, although this is a very interesting subject by itself.

Due to lack of space this paper contains only a part of the text of its *long version*, available online [18]. There we point out that developing consensus protocols is difficult and should not be undertaken in an ad-hoc manner. A resilient consensus protocol is only useful when it continues to deliver the intended service under a wide range of adversarial influence on the nodes and the network. Detailed analysis and formal argumentation are necessary to gain confidence that a protocol achieves its goal. In that sense, distributed computing protocols resemble cryptosystems and other security mechanisms; they require broad agreement on the underlying assumptions, detailed security models, formal reasoning, and wide-spread public discussion. Any claim for a “superior” consensus protocol that does not come with the necessary formal justification should be dismissed, analogously to the approach of “security by obscurity,” which is universally rejected by experts.

Section 2 reviews the role of consensus for blockchain platforms. In Section 3, we discuss the consensus protocols of a number of *permissioned* blockchain platforms, based on the available product descriptions or source code. Different consensus mechanisms not directly following the BFT approach are found in the blockchain platforms *Sawtooth Lake*, *Ripple*, *Stellar*, and *IOTA Tangle*; they are discussed in the long version. Table 1 displays a summary of the discussed protocols.

2 Consensus

This section presents background and models for consensus in permissioned blockchains, first introducing the underlying concept of state-machine replication in Section 2.1. Sections 2.2 and 2.3 briefly review the most prominent family of protocols for this task, which is based Paxos/Viewstamped Replication (VSR) and PBFT. The essential step of transaction validation is discussed in Section 2.4. In the long version [18], we furthermore demonstrate the pitfalls of consensus-protocol design, by analyzing a proposed BFT consensus protocol called *Tangaroa* and showing that it does not achieve its goals.

2.1 Blockchains and consensus

A *blockchain* is a distributed database holding a continuously growing list of records, controlled by multiple entities that may not trust each other. Records are appended to the blockchain in batches or *blocks* through a distributed protocol executed by the nodes powering the blockchain. Each block contains a cryptographic hash of the previous block, which fixes all existing blocks and embeds a secure representation of the complete chain history into every block. Additional integrity measures are often used in potentially malicious, Byzantine environments, such as the requirement that a block hash is smaller than a given target (e.g., in Nakamoto-style proof-of-work consensus), or a multi-signature (or a threshold signature) over a block, by the nodes powering the blockchain (for permissioned blockchains). The nodes communicate over a network and collaboratively construct of the blockchain without relying on a central authority.

However, individual nodes might crash, behave maliciously, act against the common goal, or the network communication may become interrupted. For delivering a continuous service, the nodes therefore run a fault-tolerant *consensus protocol* to ensure that they all agree on the order in which entries are appended to the blockchain.

Since the whole blockchain acts as a trusted system, it should be *dependable*, *resilient*, and *secure*, ensuring properties such as availability, reliability, safety, confidentiality, integrity and more [4]. A blockchain protocol ensures this by *replicating* the data and the operations over many nodes. Replication can have many roles [52, 22, 35], but blockchains replicate data only for resilience, not for scalability. All nodes validate, in principle, the information to be appended to the blockchain; this feature stimulates the trust of all nodes in that the blockchain as a whole operates correctly.

For assessing a blockchain protocol, it is important to be clear about the underlying *trust assumption* or *security model*. This specifies the environment for which the protocol is designed and in which it satisfies its guarantees. Such assumptions should cover all elements in the system, including the network, the availability of synchronized clocks, and the expected (mis-)behavior of the nodes. For instance, the typical generic trust assumption for a system with n independent nodes says that no more than $f < n/k$ nodes become *faulty* (crash, leak information, perform arbitrary actions, and so on), for some $k = 2, 3, \ldots$. The other $n - f$ nodes are *correct*. A trust assumption always represents an idealization of the real world; if some aspect not considered by the model can affect the actually deployed system, then the security must be reconsidered.

State-machine replication. The formal study and development of algorithms for exploiting replication to build resilient servers and distributed systems goes back to Lamport et al.'s pioneering work introducing Byzantine agreement [48, 38]. The topic has evolved through a long history since and is covered in many textbooks [2, 14, 49, 58]; a good summary can be found in a "30-year perspective on replication" [22].

As summarized concisely by Schneider [52], the task of reaching and maintaining consensus among distributed nodes can be described with two elements: (1) a (deterministic) *state machine* that implements the logic of the service to be replicated; and (2) a *consensus protocol* to disseminate requests among the nodes, such that each node executes the *same sequence of requests* on its instance of the service. In the literature, "consensus" means traditionally only the task of reaching agreement on one single request (i.e., the first one), whereas "atomic broadcast" [31] provides agreement on a sequence of requests, as needed for state-machine replication. But since there is a close connection between the two (a sequence of consensus instances provides atomic broadcast), the term "consensus" more often actually stands for atomic broadcast, especially in the context of blockchains. We adopt this terminology here and also use "transaction" and "request" as synonyms for one of the messages to be delivered in atomic broadcast.

Asynchronous and eventually synchronous models. Throughout this text, we assume the *eventual-synchrony* network model, introduced by Dwork et al. [26]. It models an asynchronous network that may delay messages among correct nodes arbitrarily, but eventually behaves synchronously and delivers all messages within a fixed (but unknown) time bound. Protocols in this model never violate their consistency properties (safety) during asynchronous periods, as long as the assumptions on the kind and number of faulty nodes are met. When the network stabilizes and behaves synchronously, then the nodes are guaranteed to terminate the protocol (liveness). Note that a protocol may stall during asynchronous periods; this cannot be avoided due to a fundamental discovery by Fischer et al. [27] (the celebrated "FLP impossibility result"), which rules out that deterministic protocols reach consensus in (fully) asynchronous networks.

The model is widely accepted today as realistic for designing resilient distributed systems. Replication protocols have to cope with network interruptions, node failures, system crashes, planned downtime, malicious attacks by participating nodes, and many more unpredictable effects. Developing protocols for asynchronous networks therefore provides the best possible resilience and avoids any assumptions about synchronized clocks and timely network behavior; making such assumptions can quickly turn into a vulnerability of the system if any one is not satisfied during deployment.

Protocol designers today prefer the *eventual synchrony* assumption for its simplicity and practitioners observe that it has broader coverage of actual network behavior, especially when compared to so-called partially synchronous models that assume probabilistic network behavior over time.

Consensus in blockchain. Although Nakamoto's Bitcoin paper [45] does not explicitly mention the *state-machine replication* paradigm [52], Bitcoin establishes consensus on one shared ledger based on voting among the nodes: "(Nodes) vote with their CPU power, expressing their acceptance of valid blocks by working on extending them and rejecting invalid blocks by refusing to work on them. Any needed rules and incentives can be enforced with this consensus mechanism" [45].

With the work of Garay et al. [28], a formal equivalence between the task solved by the "Nakamoto protocol" inside Bitcoin and the consensus problem in distributed computing was shown for the first time. This result coincided with the insight, developed in the fintech industry, that a blockchain platform may use a generic consensus mechanism and implement it with any protocol matching its trust model [55]. In today's understanding a blockchain platform may use an arbitrary consensus mechanism and retain most of its further aspects like distribution, cryptographic immutability, and transparency.

Existing consensus and replication mechanisms have therefore received renewed attention, for applying them to blockchain systems. Several protocols relevant for blockchains are reviewed in the next sections. We discuss only protocols for static groups here; they require explicit group reconfiguration [53, 7] and do not change membership otherwise. This assumption contrasts with view-synchronous replication [23], where the group composition may change implicitly by removing nodes perceived as unavailable.

2.2 Crash-tolerant consensus

As mentioned earlier, the form of consensus relevant for blockchain is technically known as *atomic broadcast*. It is formally obtained as an extension of a *reliable broadcast* among the node, which also provides a global or *total order* on the messages delivered to all correct nodes. An atomic broadcast is characterized by two (asynchronous) events *broadcast* and *deliver* that may occur multiple times. Every node may broadcast some message (or transaction) m by invoking *broadcast*(m), and the broadcast protocol outputs m to the local application on the node through a *deliver*(m) event.

Atomic broadcast ensures that each correct node outputs or *delivers* the same sequence of messages through the *deliver* events. More precisely [31, 14], it ensures these properties:

Validity: If a correct node p broadcasts a message m, then p eventually delivers m.

Agreement: If a message m is delivered by some correct node, then m is eventually delivered by every correct node.

Integrity: No correct node delivers the same message more than once; moreover, if a correct node delivers a message m and the sender p of m is correct, then m was previously broadcast by p.

Total order: For messages m_1 and m_2, suppose p and q are two correct nodes that deliver m_1 and m_2. Then p delivers m_1 before m_2 if and only if q delivers m_1 before m_2.

The most important and most prominent way to implement atomic broadcast (i.e., consensus) in distributed systems prone to $t < n/2$ node crashes is the family of protocols known today as *Paxos* [36, 37] and *Viewstamped Replication (VSR)* [46, 41]. Discovered independently, their core mechanisms exploit the same ideas [39, 40]. They have been implemented in dozens of mission-critical systems and power the core infrastructure of major cloud providers today [21].

The *Zab* protocol inside *ZooKeeper* is a prominent member of the protocol family; originally from Yahoo!, it is available as open source [33, 34, 56] (`https://zookeeper.apache.org/`) and used by many systems. A more recent addition to the family is *Raft* [47], a specialized variant developed with the aim of simplifying the understanding and the implementation of Paxos. It is contained in dozens of open-source tools (e.g., *etcd* – `https://github.com/coreos/etcd`).

All protocols in this family progress in a sequence of *views* or "epochs," with a unique leader for each view that is responsible for progress. If the leader fails, or more precisely, if the other nodes suspect that the leader has failed, they can replace the current leader by moving to the next view with a fresh leader. This *view change* protocol must ensure agreement, such that message already delivered by a node in the abandoned view is retained and delivered by all correct nodes in this or another future view.

2.3 Byzantine consensus

More recently, consensus protocols for tolerating *Byzantine* nodes have been developed, where nodes may be *subverted* by an adversary and act maliciously against the common goal of reaching agreement. In the eventual-synchrony model considered here, the most prominent protocol is *PBFT (Practical Byzantine Fault-Tolerance)* [19]. It can be understood as an extension of the Paxos/VSR family [39, 12, 40] and also uses a progression of views and a unique leader within every view. In a system with n nodes PBFT tolerates $f < n/3$ Byzantine nodes, which is optimal. Many research works have analyzed and improved aspects of it and made it more robust in prototypes [24].

Actual systems that implement PBFT or one of its variants are much harder to find than systems implementing Paxos/VSR. In fact, *BFT-SMaRt* (`https://github.com/bft-smart/library`) is the only known project that was developed before the interest in permissioned blockchains surged around 2015 [55]. Actually, Bessani et al. [7, 6] from the University of Lisbon started work on it around 2010. There is widespread agreement today that BFT-SMaRt is the most advanced and most widely tested implementation of a BFT consensus protocol available. Experiments have demonstrated that it can reach a throughput of about 80'000 transactions per second in a LAN [7] and very low latency overhead in a WAN [54].

Like Paxos/VSR, Byzantine consensus implemented by PBFT and BFT-SMaRt expects an eventually synchronous network to make progress. Without this assumption, only *randomized* protocols for Byzantine consensus are possible, such as the practical variations relying on distributed cryptography [16] as prototyped by SINTRA [17] or, much more recently, HoneyBadger [44].

2.4 Validation

In an atomic broadcast protocol resilient to crashes, every message is usually considered to be an acceptable request to the service. For Byzantine consensus, especially in blockchain applications, it makes sense to ask that only "valid" transactions are output by the broadcast protocol. To formalize this, the protocol is parameterized with a deterministic, external predicate $V()$, such that the protocol delivers only messages satisfying $V()$. This notion has been introduced as *external validity* by Cachin et al. [15].

The predicate must be deterministically computable locally by every process. More precisely, $V()$ must guarantee that when two correct nodes p and q in an atomic broadcast protocol have both delivered the same sequence of messages up to some point, then p obtains $V(m) = \textsc{True}$ for any message m if and only if q also determines that $V(m) = \textsc{True}$.

This combination of transaction validation and establishing consensus is inherent in permissionless blockchains based on proof-of-work consensus, such as Bitcoin and Ethereum. For permissioned-blockchain protocols, one could in principle also separate this step from consensus and perform the (deterministic) validation of transactions on the ordered "raw" sequence output by atomic broadcast. This could make the protocol susceptible to denial-of-service attacks from clients broadcasting excessively many invalid transactions. Hence most consensus protocols reviewed in this text combine ordering with validation and use a form of external validity based on the current blockchain state.

3 Permissioned blockchains

This section discusses some notable consensus protocols that are part of (or have at least been proposed for) the following consortium blockchain systems. We assume there are n nodes responsible for consensus, but some systems contain further nodes with other roles.

3.1 Overview

Among the recent flurry of blockchain-consensus protocols, many have not progressed past the stage of a paper-based description. In this section, we review only protocols *implemented* in a platform; the platform must either be available as open source or have been described in sufficient detail in marketing material. So far all implemented protocols discussed here assume independence among the failures, selfish behavior, and subversion of nodes. This

Table 1 Summary of consensus resilience properties, some of which use statically configured nodes with a *special* role. Symbols and notes: '✓' means that the protocol is resilient against the fault and '–' that it is not; '.' states that no such *special node* exists in the protocol; '?' denotes that the properties cannot be assessed due to lack of information; (✓) denotes the crash of *other* nodes, different from the special node; + MultiChain has non-final decisions; ⊕ PoET assumes trusted hardware available from only one vendor; ⊗ Ripple tolerates *one* of the five default Ripple-operated validators (special nodes) to be subverted. The last four protocols are discussed in the long version [18].

Which faults are tolerated by a protocol?	**Special-node crash**	**Any $t < n/2$ nodes crash**	**Special-node subverted**	**Any $f < n/3$ nodes subverted**
Hyperledger Fabric/Kafka	.	✓	.	–
Hyperledger Fabric/PBFT	.	✓	.	✓
Tendermint	.	✓	.	✓
Symbiont/BFT-SMaRt	.	✓	.	✓
R3 Corda/Raft	.	✓	.	–
R3 Corda/BFT-SMaRt	.	✓	.	✓
Iroha/Sumeragi (BChain)	.	✓	.	✓
Kadena/ScalableBFT	?	?	?	?
Chain/Federated Consensus	–	(✓)	–	–
Quorum/QuorumChain	–	(✓)	–	–
Quorum/Raft	.	✓	.	–
MultiChain +	.	✓	.	–
Sawtooth Lake/PoET	⊕	✓	⊕	–
Ripple	⊗	(✓)	⊗	–
Stellar/SCP	?	?	?	?
IOTA Tangle	?	?	?	?

justifies the choice of a *numeric* trust assumption, expressed only by a fraction of potentially faulty nodes.

It would be readily possible to extend such protocols to more complex fault assumptions, as formulated by *generic Byzantine quorum systems* [42]. For example, this would allow to run stake-based consensus (as done in some permissionless blockchains) or to express an arbitrary power structure formulated in a legal agreement for the consortium [11]. No platform offers this yet, however.

3.2 Hyperledger Fabric – Apache Kafka and PBFT

Hyperledger Fabric (`https://github.com/hyperledger/fabric`) is a platform for distributed ledger solutions, written in Golang and with a modular architecture that allows multiple implementations for its components. It is one of multiple blockchain frameworks hosted with the Hyperledger Project (`https://www.hyperledger.org/`) and aims at high degrees of confidentiality, resilience, flexibility, and scalability.

Following "preview" releases (*v0.5* and *v0.6*) in 2016, whose architecture [13] directly conforms to state-machine replication, a different and more elaborate design was adopted later and is currently available in release *v1.0.0-beta*. The new architecture [1], termed *Fabric V1* here, separates the execution of smart-contract transactions (in the sense of validating the inputs and outputs of a program) from ordering transactions for avoiding conflicts (in the sense of an atomic broadcast that ensures consistency). This has several advantages, including better scalability, a separation of trust assumptions for transaction validation and ordering, support for non-deterministic smart contracts, partitioning of smart-contract code and data across nodes, and using modular consensus implementations [59].

The consensus protocol up to release *v0.6-preview* was a native implementation of PBFT [19]. With V1 the *ordering service* responsible for conflict-avoidance can be provided by an *Apache Kafka* cluster (`https://kafka.apache.org/`). Kafka is a distributed streaming platform with a publish/subscribe interface, aimed at high throughput and low latency. It logically consists of *broker nodes* and *consistency nodes*, where a set of redundant brokers processes each message stream and a ZooKeeper instance (`https://zookeeper.apache.org/`) running on the consistency nodes coordinates the brokers in case of crashes or network problems. Fabric therefore inherits is basic resilience against crashes from ZooKeeper. A second implementation of the ordering service is under development, which uses again the PBFT protocol and achieves resilience against subverted nodes. Besides, BFT-SMaRt (Sec. 2.3) is currently being integrated in Fabric V1 as one of the ordering services. Since BFT-SMaRt follows the well-established literature on Byzantine consensus protocols as mentioned earlier, its properties do not need special discussion here.

3.3 Tendermint

Tendermint Core (`https://github.com/tendermint/tendermint`) is a BFT protocol that can be best described as a variant of PBFT [19], as its common-case messaging pattern is a variant of Bracha's Byzantine reliable broadcast [8]. In contrast to PBFT, where the client sends a new transaction directly to all nodes, the clients in Tendermint disseminate their transactions to the validating nodes (or, simply, validators) using a gossip protocol. The external validity condition, evaluated within the Bracha-broadcast pattern, requires that a validator receives the transactions by gossip before it can vote for inclusion of the transaction in a block, much like in PBFT.

Tendermint's most significant departure from PBFT is the continuous rotation of the leader. Namely, the leader is changed after every block, a technique first used in BFT consensus space by the *Spinning* protocol [51]. Much like Spinning, Tendermint embeds aspects of PBFT's view-change mechanism into the common-case pattern. This is reflected in the following: while a validator expects the first message in the Bracha broadcast pattern from the leader, it also waits for a timeout, which resembles the view-change timer in PBFT. However, if the timer expires, a validator continues participating in the Bracha-broadcast message pattern, but votes for a *nil* block.

Tendermint as originally described by Buchman [9] suffers from a livelock bug, pertaining to locking and unlocking votes by validators in the protocol. However, the protocol contains additional mechanisms not described in the cited report that prevent the livelock from occurring [10]. While it appears to be sound, the Tendermint protocol and its implementation are still subject to a thorough, peer-reviewed correctness analysis.

3.4 Symbiont – BFT-SMaRt

Symbiont Assembly (`https://symbiont.io/technology/assembly`) is a proprietary distributed ledger platform. The company that stands behind it, Symbiont, focuses on applications of distributed ledgers in the financial industry, providing automation for modeling and executing complex instruments among institutional market participants.

Assembly implements resilient consensus in its platform based on the open-source BFT-SMaRt toolkit (Sec. 2.3). Symbiont uses its own reimplementation of BFT-SMaRt in a different programming language; it reports performance numbers of 80'000 transactions per second (tps) using a 4-node cluster on a LAN. This matches the throughput expected from BFT-SMaRt [7] and similar results in the research literature on BFT protocols [3].

Assembly uses the standard resilience assumptions for BFT consensus in the eventually-synchronous model considered here.

3.5 R3 Corda – Raft and BFT-SMaRt

Unlike most of the other permissioned blockchain platforms discussed here, *Corda* (`https://github.com/corda/corda`) does not order all transactions as one single virtual execution that forms the blockchain. Instead, it defines *states* and *transactions*, where every transaction consumes (multiple) states and produces a new state [32]. Only nodes affected by a transaction store it. Seen across all users, this transaction execution model produces a hashed directed acyclic graph or *Hash-DAG*. Transactions must be *valid*, i.e., endorsed by the issuer and other affected nodes and correct according to the underlying smart-contract logic governing the state. Each state points to a *notary* responsible for ensuring transaction *uniqueness*, i.e., that each state is consumed only once. The notary is a logical service that can be provided jointly by multiple nodes. The *type* of a state may designate an asset represented by the network, such as a token or an obligation, or anything else controlled by a smart contract.

A transaction in Corda consumes only states controlled by the same notary; hence, one notary by itself can atomically verify the transaction's validity and uniqueness to decide whether it is executed or not. To enable transactions that operate across states governed by different notaries, there is a specialized transaction that changes the notary, such that one notary will become responsible for validating the transaction.

Since a node stores only a part of the Hash-DAG, it only knows about transactions and states that concern the node. This contrasts with most other distributed ledgers and provides means for partitioning the data among the nodes. As is the case for other smart-contract platforms, transactions refer to contracts that can be programmed in a universal general-purpose language.

A notary service in Corda orders and timestamps transactions that include states pointing to it. "Notaries are expected to be composed of multiple mutually distrusting parties who use a standard consensus algorithm" (`https://docs.corda.net`). A notary service needs to cryptographically sign its statements of transaction uniqueness, such that other nodes in the network can rely on its assertions without directly talking to the notary. Currently there is support for running a notary service as a single node (centralized), for running a distributed crash-tolerant implementation using Raft (Sec. 2.2), and for distributing it using the open-source BFT-SMaRt toolkit (Sec. 2.3). When using Raft deployed on n nodes, a Corda notary tolerates crashes of any $t < n/2$ of these nodes (Sec. 2.2). With BFT-SMaRt running on n nodes, the notary is resilient to the subversion of $f < n/3$ nodes.

3.6 Hyperledger Iroha – Sumeragi

Hyperledger Iroha (`https://github.com/hyperledger/iroha`) is another open-source blockchain platform developed under the Hyperledger Project. Its architecture is inspired by the original (v0.6) design of Fabric (Sec. 3.2). All validating nodes collaboratively execute a

Byzantine consensus protocol. In that sense it is also similar to Tendermint and Symbiont Assembly.

The *Sumeragi* consensus library of Iroha is "heavily inspired" by BChain [25] a chain-style Byzantine replication protocol that propagates transactions among the nodes with a "chain" topology. Chain replication [57, 22] arranges the n nodes linearly and each node normally only receives messages from its predecessor and sends messages to its successor. Although there is a leader at the head of the chain, like in many other protocols, the leader does not become a bottleneck since it usually communicates only with the head and the tail of the chain, but not with all n nodes. This balances the load among the nodes and lets chain-replication protocols achieve the best possible throughput [30, 3], at the cost of higher normal-case latency and slightly increased time for reconfiguration after faults.

In Sumeragi, the order of the nodes is determined based on a reputation system, which takes the "age" of a node and its past performance into account.

As becomes apparent from the documentation (`https://github.com/hyperledger/iroha/wiki/Sumeragi`), though, the protocol departs from the "chain" pattern, because the leader "broadcasts" to all nodes and so does the node at the tail. Hence, it is neither BChain nor chain replication. Assuming that Sumeragi would correctly implement BChain, then it relies on the standard assumptions for BFT consensus in the eventually-synchronous model, just like Fabric v0.6, Tendermint, and Symbiont.

3.7 Kadena – Juno and ScalableBFT

Juno from *kadena* (`https://github.com/kadena-io/juno`) is a platform for running smart contracts that has been developed until about November 2016 according to its website. Juno claims to use a "Byzantine Fault Tolerant Raft" protocol for consensus and appears to address the standard BFT model with n nodes, $f < n/3$ Byzantine faults among them, and eventual synchrony [26] as timing assumption. Later Juno has been deprecated in favor of a "proprietary BFT-consensus protocol" called *ScalableBFT* [43], which is "inspired by the Tangaroa protocol" and optimizes performance compared to Juno and Tangaroa. The whitepaper cites over 7000 transactions per second (tps) throughput on a cluster with size 256 nodes.

The design and implementation of ScalableBFT are proprietary and not available for public review. Being based on Tangaroa, the design might suffer from its devastating problems mentioned in Section 2.3. Further statements about ScalableBFT made in a blog post [50] do not enhance the trust in its safety: "Every transaction is replicated to every node. When a majority of nodes have replicated the transaction, the transaction is committed." As is well-known from the literature [22, 14] in the model considered here, with public-key cryptography for message authentication and asynchrony, agreement in a consensus protocol can only be ensured with $n > 3f$ and Byzantine quorums [42] of size *strictly larger* than $\frac{n+f}{2}$, which reduces to $2f + 1$ with $n = 3f + 1$ nodes. Hence "replicating among a majority" does *not* suffice.

The claimed performance number of more than 7000 tps is in line with the throughput of 30'000–80'000 tps, as reported by a representative state-of-the-art BFT protocol evaluation in the literature [3]. However, since Juno is proprietary, it is not not clear how it actually works nor why one should trust it, as discussed before. One should rather build on established consensus approaches and publicly validated algorithms than on a proprietary protocol for resilience. As the resilience of Juno and ScalableBFT cannot be assessed, it remains unclear whether it actually provides consensus as intended.

3.8 Chain – Federated Consensus

The *Chain Core* platform (`https://chain.com`) is a generic infrastructure for an institutional consortium to issue and transfer financial assets on permissioned blockchain networks. It focuses on the financial services industry and supports multiple different assets within the same network.

The *Federated Consensus* [20] protocol of Chain Core is executed by the n nodes that make up the network. One of the nodes is statically configured as "block generator." It periodically takes a number of new, non-executed transactions, assembles them into blocks, and submits the block for approval to "block signers." Every signer validates the block proposed for a given block height, checking the signature of the generator, validating the transactions, and verifying some real-time constraints and then signs an endorsement for the block. Each signer endorses only one block at each height. Once a node receives q such endorsements for a block, the node appends the block to its chain.

The protocol is resilient to a number of malicious (Byzantine-faulty) signers but not to a malicious block generator. If the block generator violates the protocol (e.g., by signing two different blocks for the same block height) the ledger might fork (i.e., the consensus protocol violates safety). The documentation states that such misbehavior should be addressed by retaliation and measures for this remain outside the protocol.

More specifically, when assuming the block generator operates correctly and is live, this Federated Consensus reduces to an ordinary Byzantine quorum system that tolerates f faulty signer nodes when $q = 2f + 1$ and $n = 3f + 1$; its use for consensus is similar, say, to the well-understood "authenticated echo broadcast" [14, Sec. 3.10.3]. Up to f block signers may behave arbitrarily, such as by endorsing incorrect transactions or by refusing to participate, and the protocol will remain live and available (with the correct block generator).

Overall, however, Federated Consensus is a special case of a standard BFT-consensus protocol that appears to operate with a fixed "leader" (in the role of the block generator). The protocol *cannot* prevent forks if the generator is malicious. Even if the generator simply crashes, the protocol halts and requires manual intervention. Standard BFT protocols instead will tolerate leader corruption and automatically switch to a different leader if it becomes apparent that one leader malfunctions.

Since the block generator must be correct, the purpose of a signature issued by a block signer remains unclear, at least at the level of the consensus protocol. The only reason appears to be guaranteeing that the signer cannot later repudiate having observed a block.

3.9 Quorum – QuorumChain and Raft

Quorum (`https://github.com/jpmorganchase/quorum`), mainly from developers at JP-Morgan Chase, is an enterprise-focused version of Ethereum, executing smart contracts with the Ethereum virtual machine, but using an alternative to the default proof-of-work consensus protocol of the public Ethereum blockchain. The platform currently contains two consensus protocols, called *QuorumChain* and *Raft-based consensus.*

QuorumChain. This protocol uses a smart contract to validate blocks. The trust model specifies a set of n "voter" nodes and some number of "block-maker" nodes, whose identities are known to all nodes. The documentation remains unclear about the trust model, not clearly expressing in which ways one or more of these nodes might fail or behave adversarially. (One can draw some conclusions from the protocol though.)

The protocol uses the standard peer-to-peer gossip layer of Ethereum to propagate blocks and votes on blocks, but the logic itself is formulated as a smart contract deployed with the genesis block. Nodes digitally sign every message they send. Only block-maker nodes are permitted to propose block to be appended; nodes with voter role validate blocks and express their approval by a (yes) vote. A block-maker waits for a randomly chosen time and then creates, signs, and propagates a new block that extends its own chain. A voter will validate the block (by executing its transactions and checking its consistency), “vote” on it, and propagate this. A voter apparently votes for every received block that is valid and extends its own chain, and it may vote multiple times for a given block height. Voting continues for a period specified in real time. Each node accepts and extends its own chain with the block that obtains more votes than a given threshold, and if there are multiple ones, the one with most votes. There is one block-maker node by default.

To assess the resilience of the protocol, it is obvious that already one malicious block-maker node can easily create inconsistencies (chain forks) unless the network is perfect and already provides consensus. With one block-maker, if this node crashes, the protocol halts. Depending on how the operator sets the voting threshold and on the network connectivity, it may fork the chain with only two block-makers and without any Byzantine fault. With a Byzantine fault in a block-maker node or a voter node can disrupt the protocol and also create inconsistencies. Furthermore, the protocol relies on synchronized clocks for safety and liveness. Taken together, the protocol cannot ensure consensus in any realistic sense.

Raft-based consensus. The second and more recent consensus option available for Quorum is based on the Raft protocol [47], which is a popular variant of Paxos [36] available in many open-source toolkits. Quorum uses the implementation in *etcd* (`https://github.com/coreos/etcd`) and co-locates every Quorum-node with an etcd-node (itself running Raft). Raft will replicate the transactions to all participating nodes and ensure that each node locally outputs the same sequence of transactions, despite crashes of nodes. The deployment actually tolerates that any $t < n/2$ of the n etcd-nodes may crash. Raft relies on timeliness and synchrony only for liveness, not for safety.

This is a canonical design, directly interpreting the replication of Quorum smart contracts as a replicated state machine. It seems appropriate for a protected environment, which is not subject to adversarial nodes.

3.10 MultiChain

The *MultiChain* platform (`https://github.com/MultiChain/multichain`) is intended for permissioned blockchains in the financial industry and for multi-currency exchanges in a consortium, aiming at compatibility with the Bitcoin ecosystem as much as possible.

MultiChain uses a dynamic permissioned model [29]: There is a list of permitted nodes in the network at all times, identified by their public keys. The list can be changed through transactions executed on the blockchain, but at all times, only nodes on this list validate blocks and participate in the protocol.

As the MultiChain platform is derived from Bitcoin, its consensus mechanism is called “mining” [29]; however, in the permissioned model, the nodes do not solve computational puzzles. Instead, any permitted node may generate new blocks after waiting for a random timeout, subject to a *diversity* parameter $\rho \in [0, 1]$ that constrains the acceptable miners for a given block height. More precisely, if the permitted list has length L, then a block proposal from a node is only accepted if the blockchain held by the validating node does not already contain a block generated by the *same* node among the $\lceil \rho L \rceil$ most recent blocks.

Any participating node will extend its blockchain with the first valid block of this kind that it receives, and if it learns about different, conflicting chain extensions, it will select the longer one (as in Bitcoin). Furthermore, a well-behaved node will not generate a new block if its own chain already contains a block of his within the last $\lceil \rho L \rceil$ blocks.

It appears that the random timeouts and network uncertainty easily lead to forks in the ledger, even if all nodes are correct. If two different nodes may generate a valid block at roughly the same time, and any other node will append the one of which it hears first to its chain, then these two nodes will be forked. This is not different from consensus in Bitcoin and will eventually converge to a single chain if all nodes follow the protocol. However, if a single attacking node generates transactions and blocks as it wants, and assuming that the network behaves favorably for the attack, the node can take over the entire network and revert arbitrarily many past transactions (in the same way as a "51%-attack" in Bitcoin).

Hence, MultiChain exhibits non-final transactions similar to any proof-of-work consensus. But whereas lack of finality appears to be a consequence of the public nature of proof-of-work, and since MultiChain is permissioned, forks and non-final decisions could be avoided here completely. The traditional consensus protocols for this model, discussed in Sections 2.2 and 2.3, all reach consensus with finality. In the model of non-final consensus decisions, with the corresponding delays and throughput constraints, the MultiChain consensus protocol can only remain consistent and live with one single correct node.

3.11 Further platforms

Another recent extension of the Ethereum platform is *HydraChain* (`https://github.com/HydraChain/hydrachain/blob/develop/README.md`), which adds support for creating a permissioned distributed ledger using the Ethereum infrastructure. The repository describes a proprietary consensus protocol "initially inspired by Tendermint." Without clear explanation of the protocol and formal review of its properties, its correctness remains unclear.

The *Swirlds hashgraph algorithm* is built into a proprietary "distributed consensus platform" (`https://www.swirlds.com`); a white paper is available [5] and the protocol is also implemented in an open-source consensus platform for distributed applications, called *Babble* (`https://github.com/babbleio/babble`). It targets consensus for a permissioned blockchain with n nodes and $f < n/3$ Byzantine faults among them, i.e., the standard Byzantine consensus problem according to Section 2.3. In contrast to PBFT and other protocols discussed there, it operates in a "completely asynchronous" model. The white paper states arguments for the safety and liveness of the protocol and explains that hashgraph consensus is randomized to circumvent the FLP impossibility [27]. Since the algorithm is guaranteed to reach agreement on a binary decision (i.e., with only 0/1 outcomes) only with exponentially small probability in n [5, Thm. 5.16], it appears similar to Ben-Or-style randomized agreement [14, Sec. 5.5]. However, no independent validation or analysis of hashgraph consensus is available.

Acknowledgments. We thank Andreas Kind, Pedro Moreno-Sanchez, and Björn Tackmann for interesting discussions and valuable comments.

References

1 Elli Androulaki, Christian Cachin, Konstantinos Christidis, Chet Murthy, Binh Nguyen, and Marko Vukolić. Next consensus architecture proposal. Hyperledger Wiki, Fab-

ric Design Documents, available at https://github.com/hyperledger/fabric/blob/master/proposals/r1/Next-Consensus-Architecture-Proposal.md, 2016.

2 Hagit Attiya and Jennifer Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics.* Wiley, second edition, 2004.

3 Pierre-Louis Aublin, Rachid Guerraoui, Nikola Knezevic, Vivien Quéma, and Marko Vukolic. The next 700 BFT protocols. *ACM Transactions on Computer Systems*, 32(4):12:1–12:45, 2015.

4 Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004.

5 Leemon Baird. The Swirlds hashgraph consensus algorithm: Fair, fast, Byzantine fault tolerance. Swirlds Tech Report SWIRLDS-TR-2016-01, available online, http://www.swirlds.com/developer-resources/whitepapers/, 2016.

6 Alysson Bessani and João Sousa. From Byzantine consensus to BFT state machine replication: A latency-optimal transformation. In *Proc. 9th European Dependable Computing Conference*, pages 37–48, 2012.

7 Alysson Neves Bessani, João Sousa, and Eduardo Adílio Pelinson Alchieri. State machine replication for the masses with BFT-SMaRt. In *Proc. 44th International Conference on Dependable Systems and Networks*, pages 355–362, 2014.

8 Gabriel Bracha. Asynchronous Byzantine agreement protocols. *Information and Computation*, 75:130–143, 1987.

9 Ethan Buchman. Tendermint: Byzantine fault tolerance in the age of blockchains. M.Sc. Thesis, University of Guelph, Canada, June 2016.

10 Ethan Buchman and Jae Kwon. Private discussion, 2017.

11 Christian Cachin. Distributing trust on the Internet. In *Proc. International Conference on Dependable Systems and Networks (DSN-DCCS)*, pages 183–192, 2001.

12 Christian Cachin. Yet another visit to Paxos. Research Report RZ 3754, IBM Research, November 2009.

13 Christian Cachin. Architecture of the Hyperledger blockchain fabric. Workshop on Distributed Cryptocurrencies and Consensus Ledgers (DCCL 2016), 2016. URL: https://www.zurich.ibm.com/dccl/papers/cachin_dccl.pdf.

14 Christian Cachin, Rachid Guerraoui, and Luís Rodrigues. *Introduction to Reliable and Secure Distributed Programming (Second Edition).* Springer, 2011.

15 Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. Secure and efficient asynchronous broadcast protocols (extended abstract). In Joe Kilian, editor, *Advances in Cryptology: CRYPTO 2001*, volume 2139 of *Lecture Notes in Computer Science*, pages 524–541. Springer, 2001.

16 Christian Cachin, Klaus Kursawe, and Victor Shoup. Random oracles in Constantinople: Practical asynchronous Byzantine agreement using cryptography. *Journal of Cryptology*, 18(3):219–246, 2005.

17 Christian Cachin and Jonathan A. Poritz. Secure intrusion-tolerant replication on the Internet. In *Proc. International Conference on Dependable Systems and Networks (DSN-DCCS)*, pages 167–176, June 2002.

18 Christian Cachin and Marko Vukolić. Blockchain consensus protocols in the wild. e-print, arXiv:1707.01873 [cs.DC], 2017. URL: https://arxiv.org/abs/1707.01873.

19 Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20(4):398–461, November 2002.

20 Chain protocol whitepaper. Available online, https://chain.com/docs/1.2/protocol/papers/whitepaper, 2017.

21 Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: An engineering perspective. In *Proc. 26th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 398–407, 2007.

22 Bernadette Charron-Bost, Fernando Pedone, and André Schiper, editors. *Replication: Theory and Practice*, volume 5959 of *Lecture Notes in Computer Science*. Springer, 2010.

23 Gregory V. Chockler, Idit Keidar, and Roman Vitenberg. Group communication specifications: A comprehensive study. *ACM Computing Surveys*, 33(4):427–469, 2001.

24 Allen Clement, Edmund L. Wong, Lorenzo Alvisi, Michael Dahlin, and Mirco Marchetti. Making Byzantine fault tolerant systems tolerate Byzantine faults. In *Proc. 6th Symp. Networked Systems Design and Implementation (NSDI)*, pages 153–168, 2009.

25 Sisi Duan, Hein Meling, Sean Peisert, and Haibin Zhang. BChain: Byzantine replication with high throughput and embedded reconfiguration. In Marcos K. Aguilera, Leonardo Querzoni, and Marc Shapiro, editors, *Proc. 18th Conference on Principles of Distributed Systems (OPODIS)*, volume 8878 of *Lecture Notes in Computer Science*, pages 91–106. Springer, 2014.

26 Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, 1988.

27 Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

28 Juan A. Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol: Analysis and applications. In *Advances in Cryptology: Eurocrypt 2015*, volume 9057 of *Lecture Notes in Computer Science*, pages 281–310. Springer, 2015.

29 Gideon Greenspan. Multichain private blockchain — White paper. `http://www.multichain.com/download/MultiChain-White-Paper.pdf`, 2016.

30 Rachid Guerraoui, Ron R. Levy, Bastian Pochon, and Vivien Quéma. Throughput optimal total order broadcast for cluster environments. *ACM Transactions on Computer Systems*, 28(2):5:1–5:32, 2010.

31 Vassos Hadzilacos and Sam Toueg. Fault-tolerant broadcasts and related problems. In Sape J. Mullender, editor, *Distributed Systems (2nd Ed.)*. ACM Press & Addison-Wesley, New York, 1993.

32 Mike Hearn. Corda: A distributed ledger. Available online, `https://docs.corda.net/_static/corda-technical-whitepaper.pdf`, 2016.

33 Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. ZooKeeper: Wait-free coordination for internet-scale systems. In *Proc. USENIX Annual Technical Conference*, 2010.

34 Flavio Junqueira, Benjamin Reed, and Marco Serafini. Zab: High-performance broadcast for primary-backup systems. In *Proc. 41st International Conference on Dependable Systems and Networks*, 2011.

35 Martin Kleppmann. *Designing Data-Intensive Applications*. O'Reilly Media, 2017.

36 Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.

37 Leslie Lamport. Paxos made simple. *SIGACT News*, 32(4):51–58, 2001.

38 Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

39 Butler Lampson. The ABCD's of Paxos. In *Proc. 20th ACM Symposium on Principles of Distributed Computing (PODC)*, 2001.

40 Barbara Liskov. From viewstamped replication to Byzantine fault tolerance. In Bernadette Charron-Bost, Fernando Pedone, and André Schiper, editors, *Replication: Theory and Practice*, volume 5959 of *Lecture Notes in Computer Science*, pages 121–149. Springer, 2010.

41 Barbara Liskov and James Cowling. Viewstamped replication revisited. MIT-CSAIL-TR-2012-021, July 2012.

42 Dahlia Malkhi and Michael K. Reiter. Byzantine quorum systems. *Distributed Computing*, 11(4):203–213, 1998.

43 Will Martino. Kadena — The first scalable, high performance private blockchain. Whitepaper, `http://kadena.io/docs/Kadena-ConsensusWhitePaper-Aug2016.pdf`, 2016.

44 Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The honey badger of BFT protocols. In *Proc. ACM Conference on Computer and Communications Security (CCS)*, 2016.

45 Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. Whitepaper, 2009. `http://bitcoin.org/bitcoin.pdf`.

46 Brian M. Oki and Barbara Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proc. 7th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 8–17, 1988.

47 Diego Ongaro and John K. Ousterhout. In search of an understandable consensus algorithm. In *Proc. USENIX Annual Technical Conference*, pages 305–319, 2014.

48 M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, April 1980.

49 Michel Raynal. *Communication and Agreement Abstractions for Fault-Tolerant Asynchronous Distributed Systems.* Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool, 2010.

50 George Samman. Kadena: The first real private blockchain. `http://sammantics.com/blog/2016/11/29/kadena-the-first-real-private-blockchain`, November 2016.

51 Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, and Lau Cheuk Lung. Spin one's wheels? Byzantine fault tolerance with a spinning primary. In *Proc. 28th Symposium on Reliable Distributed Systems (SRDS)*, pages 135–144, 2009.

52 Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

53 Alexander Shraer, Benjamin Reed, Dahlia Malkhi, and Flavio Paiva Junqueira. Dynamic reconfiguration of primary/backup clusters. In *Proc. USENIX Annual Technical Conference*, pages 425–437, 2012.

54 João Sousa and Alysson Bessani. Separating the WHEAT from the chaff: An empirical design for geo-replicated state machines. In *Proc. 34th Symposium on Reliable Distributed Systems (SRDS)*, pages 146–155, 2015.

55 Tim Swanson. Consensus-as-a-service: A brief report on the emergence of permissioned, distributed ledger systems. Report, available online, April 2015. URL: `http://www.ofnumbers.com/wp-content/uploads/2015/04/Permissioned-distributed-ledgers.pdf`.

56 Robbert van Renesse, Nicolas Schiper, and Fred B. Schneider. Vive la différence: Paxos vs. viewstamped replication vs. zab. *IEEE Transactions on Dependable and Secure Computing*, 12(4):472–484, 2015.

57 Robbert van Renesse and Fred B. Schneider. Chain replication for supporting high throughput and availability. In *Proc. 6th Symp. Operating Systems Design and Implementation (OSDI)*, 2004.

58 Marko Vukolić. *Quorum Systems: With Applications to Storage and Consensus.* Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool, 2012.

59 Marko Vukolić. Rethinking permissioned blockchains. In *Proc. ACM Workshop on Blockchain, Cryptocurrencies and Contracts (BCC'17)*, 2017.