From Byzantine Replication to Blockchain: Consensus is only the Beginning

Alysson Bessani*, Eduardo Alchieri†, João Sousa*, André Oliveira*, Fernando Pedone‡
*LASIGE, Faculdade de Ciências, Universidade de Lisboa, Portugal
†Departamento de Ciência da Computação, Universidade de Brasília, Brasil
‡Università della Svizzera Italiana, Lugano, Switzerland

***Abstract*—The popularization of blockchains leads to a resurgence of interest in Byzantine Fault-Tolerant (BFT) state machine replication protocols. However, much of the work on this topic focuses on the underlying consensus protocols, with particular emphasis on their lack of scalability, leaving other more subtle limitations unaddressed. These limitations are related to the effects of maintaining a durable blockchain instead of a write-ahead log and the requirement for reconfiguring the set of replicas in a decentralized way. We demonstrate these limitations using a digital coin blockchain application and BFT-SMaRt, a popular BFT replication library. We show how they can be addressed both at a conceptual level, in a protocol-agnostic way, and by implementing SMaRtChain, a blockchain platform based on BFT-SMaRt. SMaRtChain improves the performance of our digital coin application by a factor of eight when compared with a naive implementation on top of BFT-SMaRt. Moreover, SMaRtChain achieves a throughput $8\times$ and $33\times$ better than Tendermint and Hyperledger Fabric, respectively, when ensuring strong durability on its blockchain.**

I. Introduction

Recent years have seen a resurgence of interest in state machine replication (SMR) protocols, specifically in the context of permissioned blockchain systems [1]–[4]. Such protocols are used to maintain a set of stateful replicas, which execute the same set of requests in the same order, deterministically. Byzantine Fault-Tolerant (BFT) state machine replication protocols such as PBFT [5] and its descendants [6]–[11] are particularly relevant, as they implement the model properties even in the presence of an adversary that may be able to corrupt and control a fraction of the replicas. Such protocols are a direct fit for permissioned blockchains [12], where every peer/replica is known and approved to participate in the system. They are also a fundamental building block for some recent high-performance permissionless or open blockchains (e.g., [13]–[16]) that elect a subset of peers to be a transaction processing committee running the BFT protocol.

Most of the recent research on BFT replication applied to blockchain has focused on the scalability of the underlying consensus protocol [17]–[24], as most BFT protocols described before were typically designed considering few replicas. Nevertheless, there are other subtle but important differences among the BFT state machine replication approach and blockchains. One of these differences is the fact that, while many replicated state machine protocols build an internal log of executed operations for state synchronization after a leader change or a replica recovery, in a blockchain system such log must (1) be written to stable storage to ensure durability, (2) include the result of the transactions for auditing purposes, and (3) be self-verifiable by any third party. Another key difference is that while the vast majority of the literature about BFT SMR assumes a static set of processes, in a blockchain consortium, peers are expected to join and leave at any time, without the need for an additional trusted party.

In this paper, we show that these differences lead to inherent limitations, which we demonstrate by designing and running a simple digital coin blockchain application on top of BFT-SMaRt [25], a well-known BFT replication library. Our experiments show that depending on how the blockchain is implemented, and how much we are willing to trade in terms of blockchain features for better integration with the SMR library, the system throughput can go from 1.7k to 14.8k txs/sec.

Furthermore, we identify subtle issues related with *transactions persistence* and *blockchain forks*. More specifically, we show that it is possible to lose a suffix of the committed transaction history in case of a full crash of the system. This calls into question the finality of permissioned blockchains and makes them weaker in terms of durability than the centralized transactional systems they are supposed to replace. Additionally, we observe that blockchain forks might appear as a side effect of run-time consortium reconfigurations since compromised keys from past members of the consortium can be used to generate such forks.

We shown these limitations can be addressed at a conceptual level in a *protocol-agnostic way*, by describing novel mechanisms for *efficiently logging transactions and their results as a self-verifiable chain of immutable blocks* and *reconfiguring the replica set in a secure and decentralized way*. These mechanisms are independent from the consensus protocol employed to order transactions, being thus general enough to be potentially useful for any blockchain system.

The proposed techniques were implemented on SMaRtChain, a blockchain platform based on BFT-SMaRt. SMaRtChain improves the performance of the digital coin application by a factor of eight when compared with running it on top of BFT-SMaRt, and provides a performance $8\times$ and $33\times$ better than existing comparable production-level blockchains like Tendermint [3] and Hyperledger Fabric [1], respectively.

In summary, this paper makes the following contributions:

1) It identifies three fundamental limitations of running

blockchain applications on top of "classical" BFT SMR protocols: one related with potential performance issues, and two related with the gap between the state machine replication approach and blockchain requirements;
2) It introduces solutions for addressing these limitations, namely, an efficient design for transforming SMR logs in blockchains, a protocol for increasing the durability guarantee of the system, and new strategies for reconfiguring the replica set without opening breaches for blockchain forks;
3) It describes SMaRtChain, an experimental permissioned blockchain platform corresponding to the implementation of these techniques, and its evaluation showing it achieves significant performance gains when compared with similar systems.

The remainder of this paper is organized as follows. Section II presents the relevant background on blockchain and state machine replication, including BFT-SMaRt. Section III presents our system and adversary model. The gap between the SMR and blockchain is discussed in Section IV. The SMaRtChain platform is described in Section V. Section VI presents the experimental evaluation of SMaRtChain. Finally, some related works and concluding remarks are presented in Sections VII and VIII, respectively.

II. Background

A. Blockchain

The concept of blockchain was introduced by Bitcoin to solve the double spending problem associated with cryptocurrencies in open peer-to-peer networks [26]. A blockchain is an open database that maintains a distributed ledger comprised by a growing list of records called *blocks*, each of them containing transactions executed by the system. This authenticated data structure [27] consists of a sequence of blocks in which each one contains the cryptographic hash of the previous block in the chain. This ensures that block j cannot be forged without also forging all subsequent blocks $j+1...i$.

A distributed system implements a *robust transaction ledger* (i.e., a blockchain) if it satisfies the following two properties (adapted from [28]):

- *Persistence*: If a correct node reports a ledger that contains a transaction tx in a block more than k blocks away from the end of the ledger, then tx will eventually be reported in the same position in the ledger by any honest node of the system.
- *Liveness*: If a transaction is provided as input to all correct nodes, then there exists a correct node who will eventually report this transaction at a block more than k blocks away from the end of the ledger.

Blockchain systems satisfy these properties abiding to either the *permissionless* or *permissioned* models [29]. Permissionless blockchains are maintained across peer-to-peer networks in a totally decentralized and anonymous manner [26], [30]. In order to determine which block to append to the ledger, peers need to execute a Proof-of-Work (PoW) to create a valid block [28] (or an equivalent mechanism, e.g., Proof-of-Stake [19], [31]) that is disseminated to the network. The key idea behind the permissionless consensus, employed in Bitcoin and Ethereum, is to prevent an adversary from creating new blocks faster than honest participants. The first participant that finds such a solution gets to append its block to the ledger on all correct peers. Therefore, intuitively, as long as the adversary controls less than half of the total computing power present in the network, it is unable to tamper with the ledger.[1] This phenomenon also enables participants to establish a total order on the transactions by adopting the longest ledger with a valid PoW as the *de facto* transaction history.

The PoW mechanism makes permissionless blockchains slow and extremely energy demanding [29]. By contrast, permissioned blockchains do not expend as much resources and are able to reach better transaction latency and throughput. This is because nodes participating in this type of ledgers execute a traditional BFT consensus (e.g., PBFT [5]) to decide on the next block to be appended to the ledger [12]. However, this approach requires a consortium of nodes that know each other for executing the consensus protocol. In this scenario, the bound on the adversary's power is structural, not computational, i.e., safety is ensured as long as the adversary controls less than a fraction of the nodes (usually a third).

B. State Machine Replication

In the state machine replication approach [32], [33], an arbitrary number of client processes issue requests to a set of replicas. These replicas implement a stateful service that receives these requests and updates its state accordingly to the operation contained in the clients' requests. Once enough replicas transmit matching replies to the client, its invocation returns the result computed by the service.

The goal of this technique is to make the service state maintained by each replica evolve in a consistent way. In order to achieve this behavior, it is necessary to satisfy the following requirements [33]:

1) Any two correct replicas r and r' start with state s_0;
2) If any two correct replicas r and r' apply operation o to state S, both r and r' will obtain state S';
3) Any two correct replicas r and r' execute the same sequence of operations $o_0, ..., o_i$.

The first two requirements can be easily fulfilled if the service is deterministic, but the last one requires a *total order broadcast* primitive, which is equivalent to solving the *consensus problem* [34].

C. The BFT-SMaRt Library

BFT-SMaRt [25] is an open-source library that implements a modular SMR protocol [35], as well as features such as state transfer and group reconfiguration. In this section we describe these features as they are fundamental for any practical deployment of SMR.

[1] In fact the speed of the network also affects the maximum adversarial power tolerated, which is typically assumed to be much smaller than 50% [28].

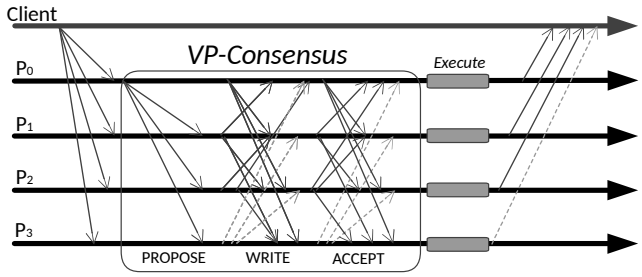


Fig. 1: BFT-SMaRt ordering message pattern.

1) SMR protocol: BFT-SMaRt uses the Mod-SMaRt protocol to implement the SMR properties described in Section II-B. Mod-SMaRt is a modular SMR protocol that works by executing a sequence of consensus instances based on the BFT consensus algorithm described in [36]. During normal operation, the resulting communication pattern is similar to the well-known PBFT protocol [5] (Figure 1). Each consensus instance i begins with a leader replica proposing a batch of client operations to be decided within that instance. All replicas that receive the proposal verify if its sender is correct by exchanging `WRITE` messages containing a cryptographic hash of the proposed batch with all other replicas. If a replica receives `WRITE` messages with the same hash from more than two thirds of the replicas, it sends a signed `ACCEPT` message to all others containing this hash. If a replica receives `ACCEPT` messages for the same hash from more than two thirds of the replicas, it delivers the corresponding batch as the decision for this consensus instance, alongside *a proof comprised by the set of signed messages* received in this last phase.

If the leader replica is faulty and/or the network experiences a period of asynchrony, Mod-SMaRt may trigger a *synchronization phase* to elect a new leader for the consensus instances and synchronize all correct replicas [35].

2) State transfer: BFT-SMaRt also allows crashed replicas to recover and resume execution. This is done by using an intermediate layer between the Mod-SMaRt protocol and the replicated service, which is responsible for triggering service checkpoints and managing the request log.

The library provides two state transfer implementations in this layer. One uses an approach similar to PBFT, that consists of storing the request log in memory, periodically truncating it after the creation of a snapshot of the service state. A recovering replica obtains the state by probing other replicas about their last completed consensus instance and asking $f+1$ replicas to send the version of the state up to that instance.[2]

The other implementation is the durability layer described in [37]. When this layer is enabled, BFT-SMaRt stores the request log into stable storage to preserve the service state even if all replicas fail by crashing. In order to write requests to disk as efficiently as possible, delivered requests are written to the durable log in parallel with their execution by the service. To better exploit the large bandwidth of stable storage devices, the system tries to write multiple batches at once, diluting the cost of a synchronous write among many requests. More specifically, the latency of writing one or ten request batches in the stable log is similar, yet the throughput would ultimately increase roughly by a factor of 10 in the latter [37].

This durability layer also enables replicas to execute checkpoints at different moments of their execution and a collaborative state transfer. These features alleviate the performance degradation caused by checkpoint generation and state transfer when the system is under heavy load.

3) Group Reconfiguration: BFT-SMaRt provides mechanisms for reconfiguring the replica set. In particular, the reconfiguration mechanism assumes the existence of a distinguished trusted client known as the *View Manager*, which uses the aforementioned state machine protocol to issue updates to the replica set. To change the current replica set (*view*) of the system, the View Manager issues a signed reconfiguration request submitted just like any other client operation. However, this request is never delivered to the application, and instead is used to update the view. Since these special operations are also totally ordered, all replicas will observe the same updates to the view along the system's lifespan.

Once the View Manager receives confirmation from the current replicas that its update was executed, it notifies the joining replicas that they can start participating in the replication protocol. At this point, they invoke the state transfer protocol to retrieve the latest application state from other replicas (as described previously) before actively participating in the replication protocol. Once these replicas receive and install the state, they are ready to process new requests.

III. System Model

We consider a fully-connected distributed system composed by a universe of processes U that can be divided in two subsets: an infinite set of replicas $\Pi = \{r_1, r_2, ...\}$, and an infinite set of clients $C = \{c_1, c_2, ...\}$. Clients access the blockchain/SMR system maintained by a subset of the replicas (a *view*) by sending their transactions to be executed and appended in the blockchain maintained by these replicas. Each process (client or server) of the system has a unique identifier. Servers and clients are prone to *Byzantine failures*. Byzantine processes are said to be *faulty*. A process that is not faulty is said to be *correct*. Each process has a *permanent public-private key pair* and has access to cryptographic functions for digital signatures and secure hashes. We assume all processes can obtain the public keys of other processes by standard means. Moreover, there are authenticated fair point-to-point links connecting every pair of processes.

We assume further an eventually synchronous system model [38]. This means the network may behave asynchronously until some unknown instant T after which it becomes synchronous, i.e., time bounds for computation and communication shall be enforced after T.

a) Dynamic replica groups: During system execution, a sequence of views is installed to account for replicas joining and leaving. Process arrivals follow the infinite arrival model

[2]In order to render this mechanism as efficient as possible, only one replica sends the entire state, while other f replicas send only a hash of it [5].

with bounded (and unknown) concurrency [39]. We assume a non-empty initial view v_{init} known to all processes (e.g., which is written in the genesis block, as will be discussed in later sections). The system *current view* cv represents the most up-to-date view installed in the system, with its replicas being the only ones that may participate in the execution of the ordering protocol. We denote by $cv.n$ the number of replicas in cv and $cv.f$ the number of replicas in cv allowed to fail, being $cv.f \leq \lfloor \frac{cv.n-1}{3} \rfloor$. A replica that asks to leave the system must remain executing the protocols until it knows that a more up-to-date view is installed, otherwise it is considered faulty.

b) Crashes and recoveries: We consider that *all replicas* in cv are subject to *recoverable crashes*, i.e., all replicas can crash at once. A replica that is in the process of being restarted is said to be in *recovery mode*, and cannot participate in the ordering protocol until its service state is restored. Therefore, the system only make progress when there are at most $cv.f$ faulty and recovering replicas.

In order to potentially bring back the entire set of replicas in cv without losing the service state, all replicas have access to a local stable storage device. Any data successfully stored in such a device will not be lost in the advent of a recoverable crash fault. Nonetheless, this guarantee does not extend to Byzantine faults, since a malicious replica is able to overwrite/corrupt its own stored data.

IV. Limitations of SMR as Blockchains

Blockchains and SMR present strong similarities since the main objective of both is to run a replicated deterministic service that executes transactions in total order. However, even if we put aside consensus protocol properties, such as finality, commit latency, and scalability [15], [29], there are still important features blockchain applications need that SMR systems do not necessarily implement. For example, blockchain applications need to maintain a self-verifiable persistent ledger with the executed transactions and support reconfigurations on the group of replicas, two features not present in most SMR implementations.

This section assesses the hindrances of the classic SMR model when supporting blockchain applications. We start by presenting an ubiquitous digital coin application used in our evaluation. Afterward, we analyze some experimental results that highlight the performance limitations of this blockchain application.

A. SMaRtCoin

To demonstrate the inherent inefficiencies of SMR for supporting blockchain applications, we developed SMaRtCoin, a digital coin application on top of BFT-SMaRt. SMaRtCoin was inspired by FabCoin, an application used to benchmark Hyperledger Fabric [1], and it represents the simplest useful blockchain application we are aware of.

SMaRtCoin is a deterministic wallet-like service that manages coins based on the UTXO (Unspent Transaction Output) model introduced in Bitcoin [26]. In this model, each object (coin) represents a certain amount of currency possessed by a user. This means that a transaction consumes a given number of input objects to produce a number of output objects. Therefore, this service supports two basic transaction types: `MINT`, used to create a certain amount of coins for a given address, and `SPEND`, to transfer coins to other addresses. The state of the service comprises a table with the coins assigned to each address in memory, and a list of addresses authorized to create new coins.

`MINT` operations require the public key of the issuer of a transaction and the value of each coin to create for the issuer. Still, it needs to have permission to execute this operation, i.e., its public key must be in the list of authorized addresses to issue `MINT` transactions, defined in the genesis block. `SPEND` operations require the issuer' public key, the id of the coins that will be used as input and a set of key-value pairs, each with an address and the amount of coins it will receive. Both types of requests need to be signed to ensure their authenticity and thus prove the ownership of the affected funds.

We implemented SMaRtCoin as a BFT-SMaRt service, using the *invoke* and *execute* interfaces provided by the library [25]. Clients generate signed SMaRtCoin transactions and submit them for the BFT-SMaRt ordering protocol. This protocol runs a Byzantine consensus to order a batch of operations, instead of a single one. Therefore, each replica receives a batch of transactions from the library's ordering protocol and delivers it to SMaRtCoin. If SMaRtCoin successfully verifies that the client that issued the transaction has the right to execute it (e.g., it is the owner of the coins being transferred), the transaction is executed.

After executing the transactions, a block containing the delivered batch together with the transactions responses is created and appended to the ledger. Once this block is synchronously written to stable storage, each replica replies to the clients with the results associated to each executed transaction.

B. SMaRtCoin Limitations

The experience of designing and running SMaRtCoin on top of BFT-SMaRt lead us to the observation of several gaps between the classic SMR and blockchain models.

a) Observation 1 (Performance issues): We run a set of experiments using different setups of SMaRtCoin on top of BFT-SMaRt. Table I reports the throughput for SMaRtCoin when writing its blockchain synchronously and asynchronously to stable storage, considering different transaction signature verification strategies. The experimental setup and methodology are detailed in Section VI. For these experiments, we configured the system with four replicas to tolerate a single Byzantine failure.

In order to compare the results with other works, it is important to consider the size of the messages exchanged since this factor significantly affects the performance of BFT protocols [25], [40]. For `MINT` operations, the requests and replies have an average size of 180 and 270 bytes, respectively. For `SPEND` operations, the size of the request is around 310 bytes, and the replies are 380 bytes long. The size of the replies

TABLE I: SMaRtCoin average throughput (txs/sec) with different signature verification and storage strategies.

Tx. type	Seq. Sign. Verification		Parallel Sign. Verification		
	sync.	async.	sync.	async.	Dura-SMaRt
MINT	1801 ± 321	1821 ± 82	4079 ± 152	4149 ± 187	15015 ± 422
SPEND	1729 ± 302	1760 ± 213	3881 ± 177	4027 ± 205	14829 ± 549

also approximates the space taken up by a serialization of each transaction (according to its type) in the ledger.

As can be seen on the left side of Table I, there is not much difference between the performance of the system with synchronous or asynchronous writes to stable storage when the signature of the coin objects is done sequentially, i.e., inside the state machine. However, if we push this verification to the BFT-SMaRt message verification pool of threads [25], effectively exploiting the multiple cores of our servers to verify signatures in parallel, we improve throughput more than twice, moving the bottleneck to the blockchain stable storage. We remark that signature verification can be further improved by parallelizing it through different replicas [41].

Although parallel signature verification significantly improves system performance, if we remove the blockchain durability implementation out of the SMR application layer, and instead use the BFT-SMaRt durability layer [37], we still have similar guarantees in terms of service durability, but the performance improves more than 3.6×. As explained in Section II-C2, this gain is due to the fact that the BFT-SMaRt durability layer accumulates several batches of transactions before delivering them to the SMR service for processing while writing these batches in a single IO operation.

b) Observation 2 (SMaRtCoin does not implement an immutable ledger): It is worth pointing out that, in all the scenarios evaluated so far, there is no immutable ledger that could be fetched to verify transactions. This happens because writing synchronously to stable storage only during the execution of the state machine, but before sending a reply to the client, ensures only what we call *external durability*: an executed operation is never reversed after the client see its completion [37]. In other words, an operation is durable only if the client that issued it receives matching replies from a (f-dissemination) Byzantine quorum with $\lfloor \frac{cv.n+cv.f+1}{2} \rfloor \geq 2cv.f+1$ replicas [42]. This ensures that these replicas wrote the operation in their logs and, even if there is a full crash and recover of the system, any other Byzantine quorum will see this operation on the log of at least one correct replica and recover the state with such operation. Notice a single log is enough because each value decided in BFT-SMaRt comes with a proof that it was the result of a consensus, as discussed in Section II-C. The consequence of this guarantee is that a single durable log of a replica does not provide a *durable committed* history of the system execution, as a suffix of the logged operations can be undone. To be sure some logged operation will not be undone, one needs to check logs from a Byzantine quorum of replicas. What is missing here is *log self-verifiability*, i.e., verifying a single correct log should be enough for obtaining the complete execution on history of the system up to that point.

c) Observation 3 (Reconfiguration depends on a centralized authority): Most BFT SMR systems assume a static set of nodes participating in the ordering protocol [5]–[11], [43], [44]. However, this is not suitable for a blockchain platform, since the set of nodes participating in the consortium are expected to change during the lifespan of the system. Moreover, there are indeed a few SMR systems that are prepared to accept new replicas to join the system and older ones to leave it, but they rely on a centralized third party with administrative privileges [45]–[47]. This is also not well suited for blockchains, since nodes should have the ability to join and leave in an autonomous way.

V. SMaRtChain

SMaRtChain is a blockchain platform based on BFT-SMaRt that efficiently support applications such as the digital coin described in Section IV-A. SMaRtChain addresses the aspects discussed in the previous section, with two novel mechanisms: the blockchain storage layer, and the decentralized reconfiguration protocol. Before diving into the details about them, we present an overview of what need to be done to transform SMR to blockchains.

A. Overview: Transforming SMR to Blockchains

The previous limitations show that naively implementing a blockchain application, even the simplest one, can result in a low-performance system with some missing features, independently of how good is the consensus being employed. Observation 1 shows that beside the scalability issues [24], [29], which have been the main focus of most of the recent work on BFT replication, it is also important to ensure that the system (1) can deal efficiently with messages of significant size, (2) is able to exploit multi-cores for cryptographic operations, and (3) implement an effective durability layer. Observations 2 and 3 are more complex to overcome and require addressing two fundamental issues on state-of-the-art SMR systems.

1) Turning Operation Logs into Blockchains: Practical SMR systems require the usage of an internal log of delivered requests, both to recover from a faulty leader and to enable the transference of service state to recovered replicas [5], [25]. The following requirements must be addressed to transform such internal log into a blockchain.

Firstly, this log must be durable. It is necessary to carefully devise a solution for log durability in order to ensure that synchronous writes to disk does not cripple the system performance [37]. Furthermore, to approach the idea of blocks, logs should no longer be comprised of individual operations, and instead composed by a sequence of blocks with the transactions ordered by the underlying protocol. Most existing SMR protocols already assume that batches of transactions

are ordered on each consensus, making the notion of blocks quite natural. In addition, each entry in the log will require a block header and a certificate that renders the block/log self-verifiable. Last, request processing and block persistence must be decoupled to ensure log self-verifiability (as defined before) and not only BFT-SMART external durability.

The second requirement is related to state snapshots. Most systems truncate the log when snapshots are created. In a blockchain platform snapshots are important as they allow a fast (re)initialization of replicas. Thus, the file in which they are stored should be linked with the chain of blocks.

Finally, the result of the transaction execution must also be stored within each block to enable auditability of transactions, matching the blockchain model.

2) Reconfiguring the Set of Nodes: As discussed before, most BFT SMR systems assume a static set of replicas, and the few that are prepared to accept replica group changes rely on a centralized third party with administrative privileges [25], [46]. Such centralized management goes against the distributed trust promised by blockchains. A more appropriate solution for a blockchain scenario would be to enable the nodes themselves to judge if some node can join the system. In addition, this mechanism should be designed in such a way that the criteria by which nodes are allowed to join should be specified by the blockchain application.

An additional problem associated with reconfigurations is how to ensure the security and verifiability of the blockchain data structure when the set of keys that validate blocks change. More specifically, new mechanisms must be designed to impede (malicious) nodes removed from the consortium to create forks on the blockchain.

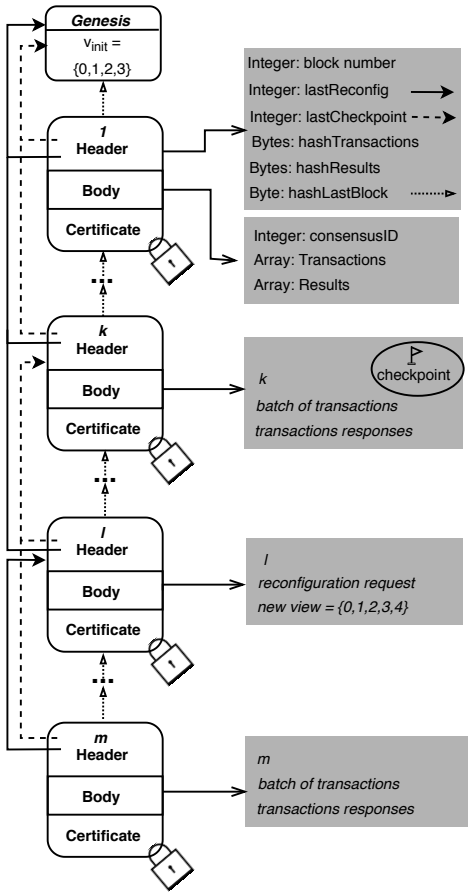


Fig. 2: SMARTCHAIN blockchain structure.

B. The Blockchain Layer

This section details how the issues previously discussed can be addressed in a blockchain design. We start by defining the blockchain data structure and then we proceed with an in-depth discussion on how it can be extended with new transactions, checkpoints, and consortium changes.

1) Blockchain structure: Figure 2 illustrates the structure of the blockchain maintained by SMARTCHAIN. On the top of the figure (block 1) we have a detailed description of a block, which is composed of three parts: (1) a header containing block metadata, (2) a body containing the list of transactions decided in a consensus instance and associated results, and (3) a certificate with a cryptographic proof of the block validity.

The header is composed of three integers representing the block number, the number of the block containing the last reconfiguration, and the block number in which the last service snapshot took place. Moreover, the header also contains hashes of the batch of transactions in the block body, the results of the execution of these transactions, and the previous block.

The body of the block contains the metadata of the consensus that delivered a batch of transactions (e.g., the consensus instance number), the list of transactions on this batch, and the list of results of each one of these transactions.[3]

The certificate comprises a set of $\lfloor \frac{cv.n+cv.f+1}{2} \rfloor \geq 2cv.f+1$ signatures of the block header generated by different replicas in the current view. In a SMR-based blockchain system this certificate suffices to guarantee that there is no other block that can be generated in this position on the blockchain.

2) Extending the Blockchain: The system starts with a genesis block containing the initial members of the consortium, their public keys, and other setup data. Every time a batch of transactions is delivered in total order and executed by the blockchain application, a new block is created containing the batch itself and the results of each transaction. This can be seen in blocks 1, k, and m in Figure 2.

3) State Checkpoints: In order to accelerate the launching of new consortium members or the time to repair crashed replicas, SMARTCHAIN still employs durable checkpoints, stored outside the blockchain. A checkpoint contains a snapshot of the application state and a reference to the *last block covered by it* (block k in Figure 2), i.e., the most recent block whose transactions were executed before the snapshot was taken. This means that a checkpoint makes the blocks before it mostly

[3]Results can include a compact representation (e.g., a Merkle tree) of the state changes caused by the transactions, making SMaRtChain compatible with execution engines like the Ethereum Virtual Machine, as in SBFT [20].

obsolete for starting a replica.

SMARTCHAIN requires a checkpoint to be created after a sequence of z blocks are processed. The parameter z is defined in the genesis block. This is different from traditional SMR systems, in which the checkpoint is defined based on the number of transactions executed. We changed it to blocks to avoid having checkpoints that partially cover a block.

Each block b stores the number c of the last block for which its transactions were included in the most recent checkpoint at the time b was created. This is important to inform anyone reading the blockchain that there is a state snapshot that represents the state of the system up to block c (inclusive).

4) Consortium Changes: A fundamental characteristic of permissioned blockchains is that members of the consortium know each other. A simple way to do that is by storing the current composition of the consortium on the blockchain.

Our blockchain structure accommodates that in two ways. First, by storing the initial consortium composition in the genesis block. Second, by storing the transaction that reconfigures the system and the corresponding new view, in a separated *reconfiguration block* (see block l in Figure 2). Just like done for checkpoints, each block stores the number of the last reconfiguration block before it in the chain. This ensures blockchain verifiers have access to enough public keys that validate the certificate of each block created in the view.

C. Strengthening the Blockchain Persistence

As discussed before, BFT-SMART provides only external durability, i.e., a transaction is irreversibly committed only if its issuer sees matching replies from a quorum of replicas (see Observation 2 in Section IV). This limitation also affects our blockchain architecture if no changes are made.

Considering the definition of blockchain in terms of Persistence and Liveness (Section II-A), this external durability is equivalent to 1-Persistence, i.e., only the second to last block is immutable. However, there are other possibilities:

- 0-*Persistence*: Perfect durability, once a block is written, it is immutable.
- α-*Persistence*: Standard durability, with α being the number of consensus instances running in parallel in the system. BFT-SMART runs a single consensus at time ($\alpha = 1$), as described before.
- λ-*Persistence*: Durability provided when using asynchronous stable storage writes. The value of λ is dependent on the environment but clearly a small integer greater than zero.
- 6-*Persistence*: The durability provided (with high probability) in the Bitcoin's blockchain [26].
- ∞-*Persistence*: No durability, provided when storing blocks only in memory.

In this paper we are particularly interested in achieving 0-*Persistence*, a guarantee similar to the durability provided by most database systems. To do that, we need an additional communication step on the system, just after the transactions are executed and persisted. This extra round of communication – designated as `PERSIST` phase – consists of making each replica generate its own signature of the block (which will now include the aforementioned transaction results) and disseminate these signatures among the view. Once a replica collects $\lfloor \frac{cv.n+cv.f+1}{2} \rfloor$ signatures for the same block, it appends these signatures to the block, thus creating a certificate for it. Notice that this write is asynchronous since if all replicas crash after synchronously writing the header and body of a block, when they recover the only possible next action is to create the same certificate again.

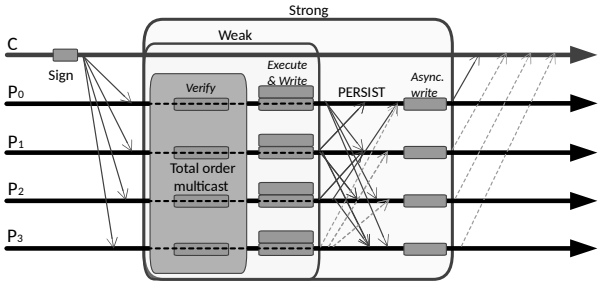


Fig. 3: SMARTCHAIN message pattern.

This modification ensures 0-Persistence because the block is considered written only when a replica knows that a Byzantine quorum of replicas executed and recorded the same set of transactions to their stable storage. Consequently, even if all replicas crash and recover, these transactions will still be visible in the blockchain.

SMARTCHAIN supports either 0- or 1-Persistence, in variants we call *weak* and *strong*, respectively. Figure 3 illustrates the message pattern of both variants. For both cases, the algorithm for state transfer is basically the same as used in BFT-SMART (Section II-C2), sending the last checkpoint covering up to a block b plus the blocks after it.

D. The Reconfiguration Protocol

SMARTCHAIN provides a new reconfiguration protocol that does not rely on a trusted third party to manage reconfigurations, allowing replicas to join/leave the system in an autonomous and secure way, following application-specific conditions.

An important aspect related with reconfigurations is how to avoid forks caused by faulty nodes removed from the system. Recall that our assumption is that in each active view v, there is at most $v.f$ faulty nodes. However, we do not assume anything about the nodes from past views. Figure 4 shows an example where the failure thresholds of all views are respected, but in which node 3, that is compromised after being removed from the system, together with faulty nodes 2 and 4 (also removed), is able to create a fork after block $k-1$ by extending the blockchain without the reconfiguration block k.

In SMARTCHAIN, we solve this problem by decoupling replicas permanent key pairs from their *consensus key pairs*, which are used to create a consensus decision proof and also to obtain a block certificate. The idea is to make all replicas generate new consensus key pairs for each view they participate, certifying each generated public consensus key

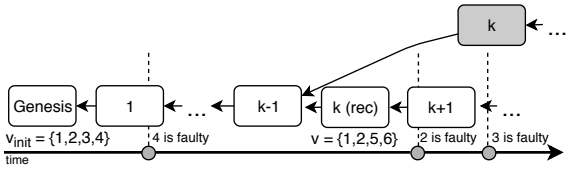


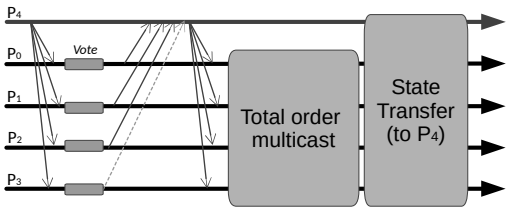
Fig. 4: Fork created by malicious processes.

with their permanent private keys, and discard their consensus key pairs on each view change. This forgetting protocol [48], [49] ensures that even if a replica becomes faulty later, after a new view is installed, it can not recover the discarded consensus private key and thus can not vouch for a block in some old view (as done by nodes 3 and 4 in the example).

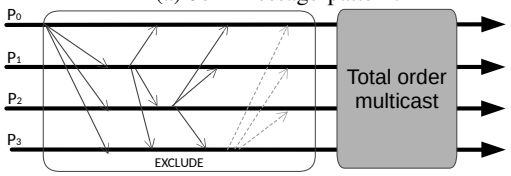
The consensus public keys for a new view need to be stored in the reconfiguration block, together with the list of members of the view. This requires the inclusion of these keys in the reconfiguration transaction. However, to preserve reconfiguration liveness in non-synchronous systems, the processes handling the reconfiguration transaction(s) that will install a new view v are ensured to collect at most $v.n - v.f$ of such keys. Fortunately, this quorum is enough for avoiding forks since, in the worst case, it will contains $v.f$ keys from faulty processes and a collusion with the $v.f$ processes whose keys were not included in the reconfiguration block (that can become malicious later) will not be enough to generate a valid proof for a consensus decision or to certify a block, which requires $\lfloor \frac{cv.n+cv.f+1}{2} \rfloor$ signatures. It is worth to mention that correct processes whose keys are not included in the reconfiguration block but that participate in the view need also to forget old keys and generate new ones. These new keys are disseminated in the first messages these processes send in the new view.

Concretely, for a new node to join the system the following steps need to be executed (Figure 5a): (1) it asks the nodes in cv for a permission to join the system; (2) each node may accept or not the request based on an application-specific policy (e.g., the new node is certified by a trusted third party, it solved a proof-of-work, or acquired a certain amount of the blockchain-specific cryptocurrency), by sending a signed reply to the joining node which also contains its new public key to be used in the next view; (3) if the joining node receives signed acceptance replies from a quorum of $cv.n - cv.f$ nodes in cv, it assembles a certificate and invokes a reconfiguration transaction that goes through the ordering protocols. After this join transaction is executed and the new node is included in the current view, its state is updated as previously described.

If a node decides to leave the system by itself, it collects public keys for a new view without itself from a quorum of nodes and notifies the others by submitting a special leave transaction in total order. Once a node receives this transaction, it generates a new view with that node excluded from the group. On the other hand, if the group decides to remove some node from the system, each node submits a special remove transaction to the ordering protocol asking for that exclusion and informing its public key for the new view (Figure 5b).



(a) Join message pattern.



(b) Exclusion message pattern.

Fig. 5: SMARTCHAIN reconfiguration protocol.

Once a node observes $cv.n - cv.f$ of such transactions from different nodes targeting the same node, it generates a new view without that node. Notice that the overhead of requiring all these transactions for running a single reconfiguration will be limited due to batching.

E. Consolidated Algorithm

Algorithm 1 consolidates all the previous ideas in a single module to be run on top of the consensus layer. During initialization, several variables are initialised and the genesis block with all consensus public keys of the initial view is written to stable storage (lines 1-10). Every time the ordering protocol delivers a batch of transactions, they are stored together with the respective consensus proofs (see Section II-C) to disk (line 18). The `asyncWriteBC` command denotes the action of asynchronously writing data to the blockchain stored in disk. Moreover, the transactions are delivered to the application code for execution (line 19) and the results are also stored to disk (line 20). This effectively creates the block's body. Since writing transactions to disk is done before executing them, asynchronously, both storage and execution are performed in parallel. Finally, the header is written to close the block (lines 21, 26-29), the replies are sent to the clients (lines 22-23), it is verified if a snapshot of the service must be created (line 24), and the blockchain becomes ready to receive the next block (line 25).

Additionally, in the strong variant, the block certificate is also created and stored in the block (lines 31-34). More specifically, each replica sends a signed `PERSIST` message with the hash of the block header to all replicas in cv. Once a replica receives correctly signed `PERSIST` messages from a quorum of replicas in cv, it creates a certificate that authenticates the block and writes it to disk.

Membership updates are stored in their own blocks (lines 37-48). The algorithm presents the processing needed to include or remove a process that asked to join or leave the system, respectively. The processing to exclude a member from the system is similar, but in this case it is necessary

Algorithm 1: SMARTCHAIN Algorithms

```
1  Upon Init do
2      myId ← replica identifier                          // replica identifier
3      bNum ← 1                                           // next block number
4      lRec ← −1                                          // last reconfiguration block number
5      lCkp ← −1                                          // last checkpoint block number
6      lbHash ← hash(∅)                                   // hash of the last block
7      lSnapshot ←⊥                                       // last state snapshot taken
8      cv ← v_init                                        // system current view
9      resetCached()                                      // resets the cached data
10     writeGenesisBlock()                                // writes the genesis block to disk

11 Procedure resetCached()
12     ∀i ∈ N : Txs[i] ← ∅                                // transactions for each block i
13     ∀i ∈ N : Res[i] ← ∅                                // responses for transactions on each block i
14     ∀i ∈ N : Cert[i] ← ∅                               // certificates for each block i
15     ∀i ∈ N : Headers[i] ← ∅                            // headers for each block i

16 Upon totalOrderDeliver ⟨BATCH, cid, txs[], proofs[]⟩ do
17     Txs[bNum] ← ⟨txs[], proofs[]⟩
18     asyncWriteBC(⟨cid, Txs[bNum]⟩)
19     Res[bNum] ← execute(Txs[bNum])
20     asyncWriteBC(Res[bNum])
21     closeBlock(⟨hash(Txs[bNum]), hash(Res[bNum])⟩)
22     foreach ⟨clientId, res⟩ ∈ Res[bNum] do
23         send ⟨REPLY, res⟩ to clientId
24     checkpoint()
25     bNum++

26 Procedure closeBlock(htx, hres)
27     Headers[bNum] ← ⟨bNum, lRec, lCkp, htx, hres, lbHash⟩
28     asyncWriteBC(Headers[bNum])
29     syncDisk()
30     lbHash ← hash(Headers[bNum])
31     if STRONG PERSISTENCE
32         send ⟨PERSIST, bNum, ⟨myId, lbHash⟩_σmyId⟩ to cv
33         wait until |Cert[bNum]| ≥ ⌈(cv.n+cv.f+1)/2⌉
34         asyncWriteBC (Cert[bNum])

35 Upon deliver ⟨PERSIST, bNum, ⟨r, lbHash⟩_σr⟩ do
36     Cert[bNum] ← Cert[bNum] ∪ {⟨r, lbHash⟩_σr}

37 Upon totalOrderDeliver ⟨VIEW, cid, recTx, recProof, nKeys[]⟩ do
38     if valid(recTx, recProof, nKeysx[])
39         Txs[bNum] ← ⟨recTx, recProof, nKeys[]⟩
40         asyncWriteBC(⟨cid, Txs[bNum], nKeys[]⟩)
41         updates cv according to recTx
42         Res[bNum] ← ⟨recTx.senderId, cv⟩
43         asyncWriteBC(Res[bNum])
44         closeBlock (hash(Txs[bNum]), hash(Res[bNum]))
45         send ⟨REPLY, cv⟩ to recTx.senderId
46         lRec ← bNum
47         checkpoint()
48         bNum++

49 Procedure checkpoint()
50     if (bNum % CHECKPOINT_PERIOD) = 0
51         lCkp ← bNum
52         resetCached()
53         lSnapshot ← takeSnapshot()
54         asyncWriteSN(lSnapshot)

55 Upon deliver ⟨ST_REQ, cid, stateReq⟩ do
56     lastTxs ← get transactions from lCkp + 1 to cid from the cache
57     send ⟨ST_REP, cid, lastTxs, lSnapshot⟩ to stateReq.senderId
```

to wait for transactions from a quorum of nodes advocating for the removal.

Finally, snapshots are written outside the blockchain in a different file (line 54) and state transfer requests are replied with the last snapshot together with the blockchain data cached since the last checkpoint (lines 55-57).

VI. EVALUATION

We implemented SMARTCHAIN over BFT-SMART and conducted several experiments *(1)* to compare the performance of different strategies for blockchain data persistence, *(2)* to compare the SMARTCHAIN performance with similar systems (Tendermint and Hyperledger Fabric), and *(3)* to understand the system behavior under events like reconfigurations, crashes, and recoveries.

A. Experimental Setup and Methodology

The experimental environment was configured with 14 machines connected to a 1Gbps switched network. The machines were configured with Ubuntu Linux 16.04.5 LTS operating system and JRE 1.8.0, hosted in Dell PowerEdge R410 servers. Each machine has 32 GB of memory and two quadcore 2.27 GHz Intel Xeon E5520 processor with hyperthreading, i.e., supporting 16 hardware threads. The machines have also a 146 GB SCSI HDD (Seagate Cheetah ST3146356SS). The experiments were conducted in up to ten replicas hosted in separate physical machines. Moreover, 2400 client processes were distributed uniformly across the other four machines.

SMARTCHAIN was configured to use a maximum batch size (block size) of 512 transactions. The experiments were conducted in two phases: the first one is composed of `MINT` operations to generate new coins, and then a second phase considers `SPEND` operations to transfer the generated coins to new addresses. Following the UTXO model, this corresponds to single-input, single-output `SPEND` transactions. Each client issued up to 1000 requests of each type (`MINT` and `SPEND`). In this section we report only the values for `SPEND` since both types of transactions present similar results.

For each experiment, the throughput was measured at the replicas at regular intervals (at each 10k operations). From the collected data, 20% of the values with greater variance were discarded and the average values are presented. Standard deviations were always under 500 txs/sec.

B. Results

This section presents the experimental results, which were divided in three subsets according to the evaluation goals.

a) Comparing different blockchain strategies: We compared the system performance considering different blockchain persistence guarantees: SMARTCHAIN configured with synchronous storage writes (0- and 1-Persistence in the strong and weak variants, respectively), asynchronous storage writes (λ-Persistence for both variants), and memory only (∞-Persistence for both variants). As a baseline, we also present results for the efficient durability layer of BFT-SMART [37], which does not implement a blockchain (Section IV-A). Figure 6 presents the throughput results for all these configurations considering different consortium sizes and the use or not of signatures.

The results show that signature verification represents the major factor that impacts performance, followed by the storage strategy. For $n = 4$ and when using signatures, SMARTCHAIN throughput reaches around 12k and 14k txs/sec for the strong

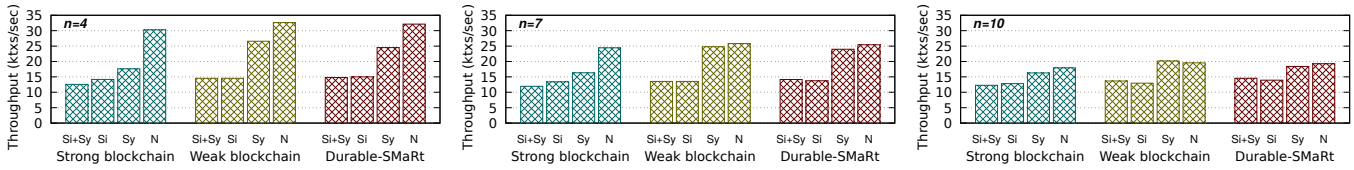


Fig. 6: SMARTCHAIN throughput for different consortium sizes and blockchain persistence guarantees. Legend: Si+Sy = Signatures and synchronous writes; Si = Signatures only; Sy = Synchronous writes only; N = None.

TABLE II: Throughput and latency for different blockchains.

Blockchain	Throughput (txs/sec)	Latency (sec)
SMARTCHAIN Strong	12560 ± 480	0.210 ± 0.033
SMARTCHAIN Weak	14547 ± 465	0.200 ± 0.023
Tendermint	1602 ± 395	1.378 ± 0.421
Hyperledger Fabric	381 ± 102	1.602 ± 0.504

and weak variants, respectively. When signatures are disabled, these values increase to around 18k and 26k txs/sec in the strong and weak variants, respectively. Notice that the size of transactions makes the throughput of plain BFT-SMART (N setup) reach 33k txs/sec, which is much less than the 80k txs/sec the system achieve with transactions of few bytes [25].

In our experiments, the size of the consortium has little impact on the performance of the configurations with stronger guarantees (signatures and synchronous writes), in all durability strategies. This shows that the consensus protocol was not the bottleneck in these scenarios. Instead, the bottleneck is the time demanded to write the ledger to disk and to perform signature verification. However, it is expected that the lack of scalability of BFT-SMART consensus protocol will make it a bottleneck in larger groups [24].

Likewise, the results show that the additional `PERSIST` phase in the strong blockchain variant does not significantly impact the performance of the system, as the obtained results for this setup are only 13% lower than the ones obtained for the weak variant.

b) Comparison with other systems: Table II compares the SMARTCHAIN performance with two other well-known BFT blockchain systems: Tendermint [3], [50], [51] and Hyperledger Fabric [1] configured with a BFT ordering service [40]. For both variants, SMARTCHAIN was configured to use signatures and synchronous writes. Both Tendermint and Hyperledger Fabric were also configured for maximum durability. Finally, all systems were configured with four replicas to tolerate a single Byzantine failure.

Table II shows that SMARTCHAIN performs significantly better than the competing systems. Tendermint uses an architecture that decouples application and ordering layers, similar to SMaRtCoin, and the performance results were also similar (Section IV-A). Although other works reported higher throughput for Hyperledger Fabric (e.g., approximately 1k txs/sec [52]), we could reach at most 381 txs/sec in our testbed.

c) Reconfigurations, crashes and recoveries: Figure 7 shows the behavior of the strong variant of SMARTCHAIN, using signatures and synchronous writes, in a run with different events and 600 clients accessing the system. For this experiment, the system was configured with 8 million UTXOs representing 10% of the current number of UTXOs in the Bitcoin network [53], leading to a state of 1GB.

We can observe that the throughput increases until all clients become operational, around second 7. At second 120, replica 4 joins the system and the throughput decreases since large quorums are used in the protocol. At second 240, replica 3 crashes, which does not impact throughput, and later recovers at second 360. In second 442, replicas perform a checkpoint that takes 23 seconds to finish. During this period the throughput drops to almost zero. It is possible to configure replicas to take checkpoints at different instants in the execution to decrease its impact in the overall system performance [37]. Finally, at second 480, replica 4 leaves the system and throughput goes back to what was observed in the beginning of the experiment.

Notice that after a join or a recovery, replicas demand approximately 60 seconds to obtain and install the 1GB state from the other replicas (green spots in Figure 7). Throughput is slightly smaller during this period since replicas must send their state to the joining/recovering replica. By using checkpoints and state transfer, a replica is able to join the system faster than in other systems that do not employ this technique. For example, currently a node must process a blockchain of 223GB (9080186 blocks) to join the Ethereum network [54], even pruning old states. Based on this observation, Figure 8 shows the processing time demanded to update a replica considering different checkpoint periods and blockchain sizes. Checkpoints boost the reconfiguration performance since joining nodes need to process only the transactions logged after the last checkpoint.

VII. RELATED WORK

Since Bitcoin's inception and widespread adoption there have been an impressive amount of work on both permissionless and permissioned blockchain platforms. Most of these works focus on the multiple types of blockchain consensus, but very few provide an in-depth discussion about blockchain durability and the issues with decentralized consortium reconfiguration.

a) Durability: The scale, latency, and probabilistic finality of the most popular blockchains lead to an ad-hoc

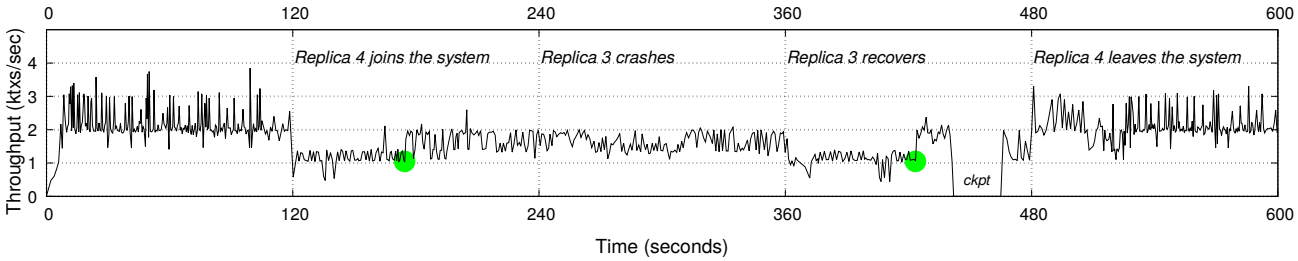


Fig. 7: Throughput evolution across time and events, $v_{init} = \{0, 1, 2, 3\}$.

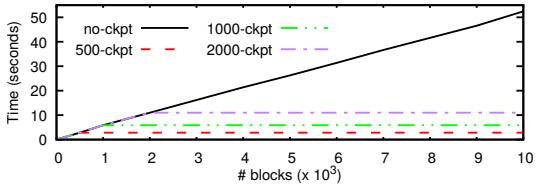


Fig. 8: Time demanded to update a replica.

implementation of blockchain durability. However, the recent popularization of small-scale permissioned blockchains (e.g., [1]–[4], [55]) and their use as distributed transaction platforms [56], [57], calls for a better understanding of blockchain durability. However, to the best of our knowledge, this subject was not yet explored in both academic and industrial works.

One of the best known blockchain platforms is Hyperledger Fabric [1]. The platform is designed to support pluggable implementations of different components, such as the ordering and membership services. Fabric's key innovation is the execution of transactions before establishing a total order among (blocks of) them. Only after such order is established the blocks are validated by the peers and then written to stable storage. Fabric durability guarantees are not well documented, but the lack of coordination between peers during blockchain writing suggest that the system offers guarantees at most like SMARTCHAIN weak persistence.

Tendermint is another notorious permissioned platform that implements a variant of the PBFT protocol [3], making its design more similar to SMARTCHAIN than Fabric. However, Tendermint has two distinguished features: it uses a gossip protocol to propagate transactions among nodes, and it adopts a leader rotation mechanism similar to Spinning [9]. In terms of persistence, Tendermint writes the block before and after operation execution, making it less efficient than SMARTCHAIN (as evidenced by our experimental results), without further coordination between the replicas. The consequence is that the system supports only weak persistence for its blockchain.

b) Consortium reconfiguration: Some works have also tackled the challenges of supporting group reconfiguration in SMR [13], [15], [45]–[47]. ComChain [58], Hybrid Consensus [13], and Solida [15] are the ones that most resemble our solution since they support fully autonomous reconfiguration. Similarly to our approach, ComChain allows reconfigurations to be defined by application-specific criteria but does not deal with forks. Hybrid Consensus determines the committee members using Bitcoin's (PoW-based) protocol while using a traditional consensus protocol among current committee members to order transactions. On the other hand, our solution is entirely derived from a classic BFT state machine protocol. Moreover, Solida is designed to operate in the synchronous system model and uses a variant of the PBFT protocol adapted to such model. Our solution is still able to operate in an eventually synchronous model, like most SMR protocols in the literature.

Fabric and Tendermint also support consortium reconfigurations. Fabric only allows reconfiguration with the help of a trusted network administrator [59]. Tendermint, in principle, supports decentralized reconfigurations if the application defines how this should be done [60]. However, none of these systems deal with the potential forks that might arise with multiple reconfigurations.

VIII. CONCLUSIONS

This paper discussed some misalignments between the state machine replication approach and the permissioned blockchain requirements and proposed several techniques to address them. The identified issues concern the low performance of blockchain applications, the lack of strong blockchain persistence guarantees, and the possibility of forks in consortium reconfigurations. We propose a set of consensus-agnostic techniques materialized in a blockchain layer that can be integrated into SMR frameworks to mitigate these issues. To validate our approach, we implemented these techniques on SMARTCHAIN, a proof-of-concept permissioned blockchain on top of BFT-SMART. Experimental results show that SMARTCHAIN improves the performance of a simple digital coin application by 8× when compared with running it on top of BFT-SMART, and by 8× and 33× when compared with Tendermint and Hyperledger Fabric, respectively.

Acknowledgements: We thank Michael Davidson, Vincent Gramoli, Dragos-Adrian Seredinschi, the anonymous reviewers, and our shepherd, Heming Cui, for their comments that significantly improved this paper. This work was supported by FCT through projects IRCoC (PTDC/EEI-SCR/6970/2014), ThreatAdapt (FCT-FNR/0002/2018), and the LASIGE Research Unit (UIDB/00408/2020 and UIDP/00408/2020), by FAPDF/Brazil through Edital 05/2018, and by the Swiss National Science Foundation (project number 175717).

REFERENCES

[1] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. D. Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, S. Muralidharan, C. Murthy, B. Nguyen, M. Sethi, G. Singh, K. Smith, A. Sorniotti, C. Stathakopoulou, M. Vukolic, S. W. Cocco, and J. Yellick, "Hyperledger fabric: A distributed operating system for permissioned blockchains," in *Proceedings of the 13th ACM SIGOPS European Conference on Computer Systems*, 2018.

[2] "Chain protocol whitepaper," 2014. [Online]. Available: https://chain.com/docs/1.2/protocol/papers/whitepaper

[3] E. Buchman, "Tendermint: Byzantine fault tolerance in the age of blockchains," Master's thesis, University of Guelph, 2016.

[4] W. Martino, "Kadena: The first scalable, high performance private blockchain," 2016. [Online]. Available: http://kadena.io/docs/Kadena-ConsensusWhitePaper-Aug2016.pdf

[5] M. Castro and B. Liskov, "Practical Byzantine fault tolerance," in *Proc. of the USENIX Symposium on Operating Systems Design and Implementation*, 1999.

[6] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira, "HQ-Replication: A hybrid quorum protocol for Byzantine fault tolerance," in *Proceedings of 7th Symposium on Operating Systems Design and Implementation - OSDI 2006*, Seattle, Washington, Nov. 2006.

[7] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, "Zyzzyva: Speculative Byzantine fault tolerance," in *Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles*, 2007.

[8] Y. Amir, B. Coan, J. Kirsch, and J. Lane, "Prime: Byzantine replication under attack," *IEEE Transactions on Dependable and Secure Computing*, vol. 8, no. 4, pp. 564–577, 2011.

[9] G. S. Veronese, M. Correia, A. N. Bessani, and L. C. Lung, "Spin one's wheels? Byzantine fault tolerance with a spinning primary," in *Proceedings of the 28th IEEE Symposium on Reliable Distributed Systems*, Niagara Falls, NY, USA, Sep. 2009.

[10] G. Veronese, M. Correia, A. Bessani, L. C. Lung, and P. Verissimo, "Efficient Byzantine fault-tolerance," *IEEE Transactions on Computers*, vol. 62, no. 1, pp. 16–30, 2013.

[11] P.-L. Aublin, S. B. Mokhtar, and V. Quéma, "RBFT: Redundant Byzantine fault tolerance," in *Proceedings of the 2013 IEEE 33rd International Conference on Distributed Computing Systems*, Philadelphia, PA, USA, 2013.

[12] C. Cachin and M. Vukolic, "Blockchain consensus protocol in the wild (invited paper)," in *Proceedings of 31th International Symposium on Distributed Computing*, Vienna, Austria, 2017.

[13] R. Pass and E. Shi, "Hybrid Consensus: Efficient Consensus in the Permissionless Model," in *Proceedings of the 31st International Symposium on Distributed Computing (DISC 2017)*, 2017, pp. 39:1–39:16.

[14] E. Kokoris-Kogias, P. Jovanovic, N. Gailly, I. Khoffi, L. Gasser, and B. Ford, "Enhancing bitcoin security and performance with strong consistency via collective signing," in *Proceedings of the 25th USENIX Conference on Security Symposium*, ser. SEC'16. USENIX Association, 2016, pp. 279–296.

[15] I. Abraham, D. Malkhi, K. Nayak, L. Ren, and A. Spiegelman, "Solida: A Blockchain Protocol Based on Reconfigurable Byzantine Consensus," in *21st International Conference on Principles of Distributed Systems (OPODIS 2017)*, 2017, pp. 25:1–25:19.

[16] J. Yu, D. Kozhaya, J. Decouchant, and P. Esteves-Verissimo, "Repucoin: Your reputation is your power," *IEEE Transactions on Computers*, vol. 68, no. 8, pp. 1225–1237, 2019.

[17] A. Miller, Y. Xia, K. Croman, E. Shi, and D. Song, "The honey badger of BFT protocols," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: ACM, 2016, pp. 31–42. [Online]. Available: http://doi.acm.org/10.1145/2976749.2978399

[18] S. Duan, M. K. Reiter, and H. Zhang, "BEAT: Asynchronous BFT made practical," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. ACM, 2018, pp. 2028–2041.

[19] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich, "Algorand: Scaling byzantine agreements for cryptocurrencies," in *Proceedings of the 26th Symposium on Operating Systems Principles*, ser. SOSP '17, 2017, pp. 51–68.

[20] G. Golan-Gueta, I. Abraham, S. Grossman, D. Malkhi, B. Pinkas, M. K. Reiter, D. Seredinschi, O. Tamir, and A. Tomescu, "SBFT: a scalable decentralized trust infrastructure for blockchains," in *Proc. of the 19th IEEE/IFIP Int. Conf. on Dependable Systems and Networks – DSN'19*, 2019.

[21] Y. Yang, "Linbft: Linear-communication byzantine fault tolerance for public blockchains," *CoRR*, vol. abs/1807.01829, 2018. [Online]. Available: https://arxiv.org/abs/1807.01829

[22] J. Liu, W. Li, G. O. Karame, and N. Asokan, "Scalable byzantine consensus via hardware-assisted secret sharing," *IEEE Transactions on Computers*, vol. 68, no. 1, 2019.

[23] T. Crain, V. Gramoli, M. Larrea, and M. Raynal, "DBFT: Efficient leaderless Byzantine consensus and its application to blockchains," in *2018 IEEE 17th International Symposium on Network Computing and Applications (NCA)*, Nov 2018, pp. 1–8.

[24] R. Guerraoui, J. Hamza, D.-A. Seredinschi, and M. Vukolic, "Can 100 machines agree?" *CoRR*, vol. abs/1911.07966, 2019. [Online]. Available: http://arxiv.org/abs/1911.07966

[25] A. Bessani, J. Sousa, and E. Alchieri, "State machine replication for the masses with BFT-SMART," in *Proceedings of the 44th IEEE/IFIP International Conference on Dependable Systems and Networks*, 2014.

[26] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2009. [Online]. Available: http://bitcoin.org/bitcoin.pdf

[27] R. Tamassia, "Authenticated data structures," in *Proceedings of European Symposium on Algorithms*, 2003.

[28] J. Garay, A. Kiayias, and N. Leonardos, "The bitcoin backbone protocol: Analysis and applications," in *Proceedings of the 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques*, Sofia, Bulgaria, 2015.

[29] M. Vukolić, "The quest for scalable blockchain fabric: Proof-of-work vs. BFT replication," in *Open Problems in Network Security - IFIP WG 11.4 International Workshop*, Zurich, Switzerland, 2015.

[30] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," 2015. [Online]. Available: http://gavwood.com/Paper.pdf

[31] A. Kiayias, A. Russell, B. David, and R. Oliynykov, "Ouroboros: A provably secure proof-of-stake blockchain protocol," in *Advances in Cryptology – CRYPTO 2017*, J. Katz and H. Shacham, Eds. Cham: Springer International Publishing, 2017, pp. 357–388.

[32] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[33] F. Schneider, "Implementing fault-tolerant service using the state machine aproach: A tutorial," *ACM Computing Surveys*, vol. 22, no. 4, pp. 299–319, 1990.

[34] V. Hadzilacos and S. Toueg, "Fault-tolerant broadcasts and related problems," in *Distributed Systems (2nd Ed.)*, S. Mullender, Ed. ACM Press/Addison-Wesley Publishing Co., 1993, pp. 97–145. [Online]. Available: http://dl.acm.org/citation.cfm?id=302430.302435

[35] J. Sousa and A. Bessani, "From Byzantine consensus to BFT state machine replication: A latency-optimal transformation," in *Proceedings of the 9th European Dependable Computing Conference*, 2012.

[36] C. Cachin, "Yet another visit to Paxos," IBM Research Zurich, Tech. Rep. RZ 3754, 2009.

[37] A. Bessani, M. Santos, J. Felix, N. Neves, and M. Correia, "On the efficiency of durable state machine replication," in *Proceedings of the 2013 USENIX Annual Technical Conference*, San Jose, CA, USA, 2013.

[38] C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the presence of partial synchrony," *J. ACM*, vol. 35, no. 2, pp. 288–323, Apr. 1988. [Online]. Available: http://doi.acm.org/10.1145/42282.42283

[39] M. K. Aguilera, "A pleasant stroll through the land of infinitely many creatures," *SIGACT News*, vol. 35, no. 2, pp. 36–59, Jun. 2004. [Online]. Available: http://doi.acm.org/10.1145/992287.992298

[40] J. Sousa, A. Bessani, and M. Vukolic, "A byzantine fault-tolerant ordering service for the hyperledger fabric blockchain platform," in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, June 2018, pp. 51–58.

[41] T. Crain, C. Natoli, and V. Gramoli, "Evaluating the Red Belly blockchain," *CoRR*, vol. abs/1812.11747, 2018. [Online]. Available: http://arxiv.org/abs/1812.11747

[42] D. Malkhi and M. Reiter, "Byzantine quorum systems," *Distributed Computing*, vol. 11, no. 4, pp. 203–213, 1998.

[43] M. Abd-El-Malek, G. Ganger, G. Goodson, M. Reiter, and J. Wylie, "Fault-scalable Byzantine fault-tolerant services," in *Proceedings of the 20th ACM SIGOPS Symposium on Operating Systems Principles*, Brighton, UK, 2005.

[44] J. Behl, T. Distler, and R. Kapitza, "Hybrids on steroids: SGX-based high performance BFT," in *Proceedings of the 12th ACM SIGOPS European Conference on Computer Systems*, 2017.

[45] J. R. Lorch, A. Adya, W. J. Bolosky, R. Chaiken, J. R. Douceur, J. Howell, J. J. Douceur, J. Lorch, and B. Bolosky, "The SMART way to migrate replicated stateful services," in *Proceedings of the 2006 ACM/SIGOPS EuroSys Conference*, April 2006.

[46] R. Rodrigues, B. Liskov, K. Chen, M. Liskov, and D. Schultz, "Automatic reconfiguration for large-scale reliable storage systems," *IEEE Transactions on Dependable and Secure Computing*, vol. 9, no. 2, pp. 145–158, March 2012.

[47] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *2014 USENIX Annual Technical Conference*, Philadelphia, PA, USA, 2014.

[48] J.-P. Martin and L. Alvisi, "A framework for dynamic Byzantine storage," in *Proceedings of the International Conference on Dependable Systems and Networks*, 2004.

[49] R. Rodrigues and B. Liskov, "Rosebud: A scalable Byzantine-fault-tolerant storage architecture," MIT Laboratory for Computer Science, MIT-LCS-TR 932, 2004.

[50] Y. Amoussou-Guenou, A. D. Pozzo, M. Potop-Butucaru, and S. Tucci-Piergiovanni, "Correctness of Tendermint-Core Blockchains," in *Proceedings of the 22nd International Conference on Principles of Distributed Systems (OPODIS 2018)*, 2018, pp. 16:1–16:16.

[51] E. Buchman, J. Kwon, and Z. Milosevic, "The latest gossip on BFT consensus," *CoRR*, vol. abs/1807.04938, 2018. [Online]. Available: http://arxiv.org/abs/1807.04938

[52] S. Rusch, K. Bleeke, and R. Kapitza, "Bloxy: Providing transparent and generic BFT-based ordering services for blockchains," in *Proceedings of the 38th IEEE Symposium on Reliable Distributed Systems*, 2019.

[53] B. Chart, "Number of unspent transaction outputs," 2019. [Online]. Available: https://www.blockchain.com/charts/utxo-count

[54] Etherscan, "Ethereum full node sync default chart," 2019. [Online]. Available: https://etherscan.io/chartsync/chaindefault

[55] D. Voell and P. M. Nielsen, "Quorum whitepaper," 2016. [Online]. Available: https://github.com/jpmorganchase/quorum-docs/blob/master/Quorum%20Whitepaper%20v0.1.pdf

[56] S. Nathan, C. Govindarajan, A. Saraf, M. Sethi, and P. Jayachandran, "Blockchain meets database: Design and implementation of a blockchain relational database," *Proc. VLDB Endow.*, vol. 12, no. 11, pp. 1539–1552, Jul. 2019.

[57] A. Sharma, F. M. Schuhknecht, D. Agrawal, and J. Dittrich, "Blurring the lines between blockchains and database systems: The case of hyperledger fabric," in *Proceedings of the 2019 International Conference on Management of Data*, ser. SIGMOD '19, 2019, pp. 105–122.

[58] G. Vizier and V. Gramoli, "ComChain: A blockchain with Byzantine fault-tolerant reconfiguration," *Concurrency and Computation: Practice and Experience*, 2019, available online.

[59] Fabric Documentation, https://hyperledger-fabric.readthedocs.io/en/release-1.4/config_update.html.

[60] Tendermint Documentation, https://godoc.org/github.com/tendermint/tendermint/lite#hdr-How_We_Track_Validators.