IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received July 30, 2020, accepted August 24, 2020, date of publication August 31, 2020, date of current version September 11, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3020319

ZyConChain: A Scalable Blockchain for General Applications

NASRIN SOHRABI AND ZAHIR TARI
School of Science, RMIT University, Melbourne, VIC 3001, Australia

Corresponding author: Nasrin Sohrabi (s3732890@student.rmit.edu.au)

This work was supported by the Australian Research Council (ARC) under the Discovery Scheme under Grant DP200100005.

ABSTRACT Blockchain's popularity has seen a historic rise over the last decade. However, existing blockchain systems have a major issue with *scalability*, which has become one of the main obstacles in technology's adoption in mainstream. There have been several attempts to address this limitation by identifying Blockchain's scalability/performance bottlenecks (e.g. those mainly related to consensus algorithms), and thus proposed different solutions (e.g., new consensus protocols) to address such limitations. Other works applied *sharding* to tackle the issue. All solutions however have mainly focused on Cryptocurrency applications, and thus addressing the scalability of blockchain systems for general applications remains a concern. This work proposes a scalable blockchain protocol for general applications (i.e., not restricted to Cryptocurrencies). To improve the two major factors affecting transaction scalability, namely *throughput* and *latency*, we needed to modify both the blockchain structure as well as the block generation process. ZyConChain, the proposed Blockchain system, introduces three types of blocks that form three separate chains: *parentBlock*, *sideBlock* and *state block*. These blocks are generated based on different consensus algorithms, as each algorithm has specific properties that make it suitable for each type of block. To improve the overall performance, ZyConChain generates sideBlocks (that carry transactions) at a high rate and keep them in a pool. To generate parentBlock, miners, instead of packing transactions into a block as they do in conventional blockchains, pack sideBlocks into a parentBlock. SideBlocks are generated based on an adapted Zyzzyva consensus protocol, with $O(\log n)$ complexity. This has reduced the final consensus complexity per transaction, in comparison to previous work. To enable the protocol to scale out with the increase in the number of nodes, ZyConChain applied sharding technique. Parallel state chains have also been introduced to address cross-shard transactions.

INDEX TERMS Blockchain, consensus protocol, distributed systems, scalability, security, sharding.

I. INTRODUCTION

Blockchain, as a promising solution to develop secure distributed ledgers, has gained an increasing attention over the last decade. It is an append-only data structure that runs over a decentralized network and is distributed among all the peers. The strong security features, namely *integrity, immutability*, and *transparency* that Blockchain offers, has made the technology well-suited for applications with high security requirements, and can benefit other sectors, such as Supply chain, e-voting, Internet of Things (IoT), Healthcare & medical records, and Government records (e.g., date of birth). However, to provide these features (*decentralization* and *security*), blockchain has jeopardized *scalability* [1]. When dealing with an ever increasing number of users, miners, and transactions, blockchain is unable to scale and provide the same performance as centralised systems (e.g. centralised payment systems). Without addressing this fundamental scalability problem, such a promising technology may not be able to be adopted in mainstream.

Several solutions have been proposed to address scalability, such as Bitcoin-NG [2], ByzCoin [3], Elastico [4], Omniledger [5], RapidChain [6], Red Belly [7], Algorand [8], IoTA [9], Directed Acyclic Graph (DAG) based solutions (e.g., Byteball [10], nano [11]), and Ripple [12], [13]. Some of these proposals have achieved a better performance by modifying blockchain structure. However, others identified the consensus algorithm as the major bottleneck of scalability,

The associate editor coordinating the review of this manuscript and approving it for publication was Tony Thomas.

and hence, they modified the consensus algorithm to resolve the issue. But these approaches have some major drawbacks, and here we summarise some of these limitations and show the proposed Zyzzyva algorithm addresses them.

A. SUMMARY OF EXISTING SOLUTIONS

ByzCoin [3] focused on designing a blockchain consensus protocol to reduce the transactions' latency of Bitcoin [14], [15]. Inspired by Bitcoin-NG [16], ByzCoin decouples transactions from the block generation and divides the conventional block into two blocks, called KeyBlock and microBlock. The former relates to the election of a leader among the consensus group, and the latter form the transactions blocks. By decoupling the leader election and transaction block, ByzCoin was able to improve throughput and latency. However, scalability remains a problem in ByzCoin and Bitcoin-NG, as they are unable to scale out when the number of miners in the network increases [17]. Another well-studied approach to address scalability is the sharding method [4], which partitions the nodes in a network into small groups (called *committees*) with the aim that committees work in parallel to process transactions. This method has been applied in several works, such as Elastico [4], Omniledger [5], and RapidChain [6]. However, these still have limitations, especially when applying sharding, where cross-shard transactions processing needs to be addressed. Cross-shard transaction refers to the transaction that results in the update of two or more shards. The existing sharding-based methods either fail to process cross-shard transactions (e.g., Elastico [4]) or their applicability is limited to Cryptocurrency applications (e.g., Omniledger [5] and RapidChain [6]). Thus, they cannot be applied for general applications.

B. CONTRIBUTIONS' SUMMARY

The aim of this article is to build a scalable Blockchain protocol that can be applied to all sectors with different applications. The focus is (i) to increase throughput, (ii) to reduce latency, and (iii) to enable the protocol to scale out in proportion to the number of nodes.

- To achieve (i)-(ii), we modified the block structures, block generation process, and the consensus algorithm. We introduced three types of blocks (i.e., *parentBlock*, *sideBlock*, and *state block*) and they form different chains. Thus, the protocol comprises of three different chains: *main chain* (which comprises of parentBlocks), *sideBlock chain*, and *state chain*. The blocks are generated in different layers with different consensus algorithms. ParentBlock generation follows the Nakamoto consensus model. For the sideBlock generation however, we introduced a new consensus protocol based on Zyzzyva consensus algorithm [18]. Zyzzyva is a Byzantine Fault Tolerance (BFT) consensus protocol that enhances the performance of the previous BFT consensus protocols, such as Practical Byzantine Fault Tolerance (PBFT) [19]. Similar to other BFT consensus algorithms, Zyzzyva can't scale up. To address this issue of scalability, we applied the Scalable Collective Signature (CoSi) protocol [20].
- To achieve (iii), we applied sharding method, and the innovation here is on the way we addressed the cross-shard transactions: parallel chains, which comprise of the state chains of all shards in the system, are introduced. Having the state chains of other shards, which contain information about cross-shard transactions, enable the nodes to verify the cross-shard transactions.

This new proposed protocol is called *ZyConChain*. To generate sideBlock, we choose Zyzzyva algorithm [18], after conducting an evaluation of consensus protocols in distributed systems (for details refer to Appendix A). The performance analysis provided in [18], [21] showed that Zyzzyva has significantly improved performance compared to the previous BFT protocols, namely, PBFT [19] and QU [22]. Zyzzyva's latency has reached the lower bound and its throughput overhead has reduced remarkably. We believe this fundamental improvement can notably benefit blockchain protocols if applied correctly. Thus, Zyzzyva is used here to introduce a new blockchain consensus protocol to reduce the latency and increase the throughput of current protocols.

Zyzzyva applied a speculative approach in updating the states, resulted in a high throughput [21], [23]. When the primary (current leader) receives a client's request, it sends it to other nodes. Nodes respond to the request without first running the expensive three-phase commit protocol to reach agreement. It comprises of two paths [18], [21], [23]: one is a *two-phase* path (that resembles the PBFT protocol), and the other is the *fast* path (that has removed the *commit* phase from the PBFT protocol). In the fast path, the state is updated once the client receives $3f + 1$ *prepare* message. However, if there is not enough $(3f + 1)$ *prepare* messages, then the protocol falls into the two-phase path to guarantee the progress. Below are some details about the two paths:

- *Fast path* does not have commit message. When a client receives the $3f + 1$ *prepare* messages from the nodes, it commits the message. This is an optimistic approach that can fail. To guarantee the progress, Zyzzyva proposes a second path, *two-phase* path which resembles the PBFT.
- *Two-phase path*: if the client receives between $2f + 1$ and $3f$ *prepare* messages, then the client needs to collect $2f + 1$ *commit* messages. Thus, the client creates a *commit-certificate* and sends it to the nodes. The nodes send the *commit* message to the client. If client receives $2f + 1$ *commit* messages, then the request is complete and client commits the message.

To evaluate the performance of *ZyConChain* protocol, its consensus complexity per transaction is compared with existing ones. The results show that ZyConChain outperforms the previous sharding-based blockchain protocols. To measure the overheads of the parallel state chains that have been added

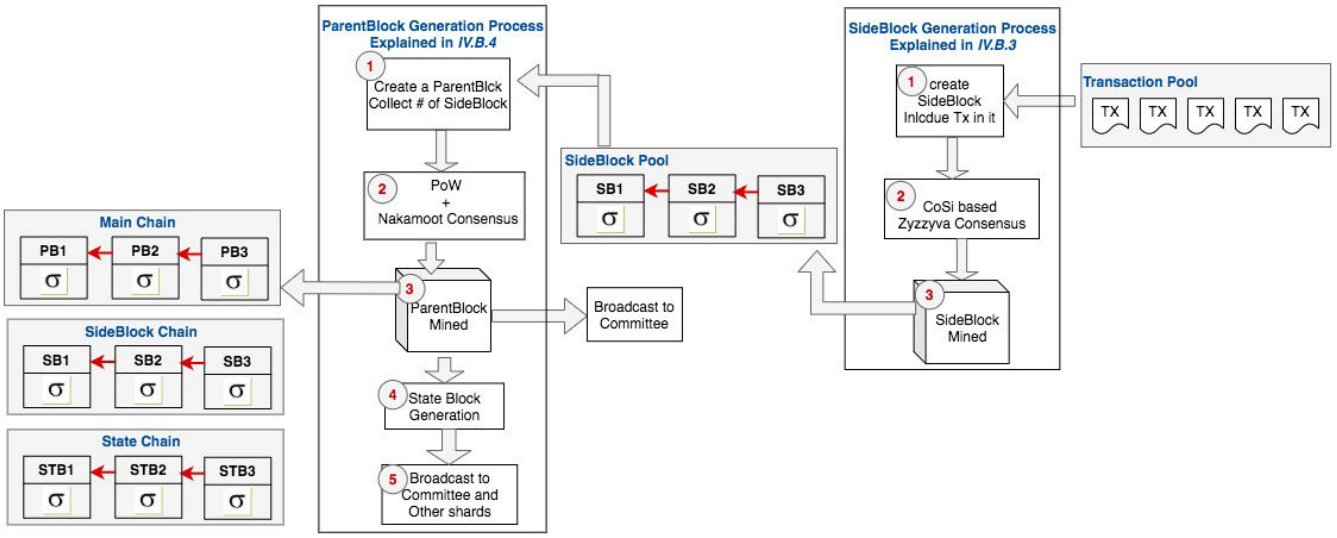

FIGURE 1. ZyConChain high level view.

to the protocol, we measured the storage complexity and performed a comparison with other works. The results also show that the overheads incurred by the parallel chains is negligible.

C. ZyConChain's OVERVIEW

Figure 1 provides a high-level view of ZyConChain. This divides the network into small groups, *committee*, and introduces three types of chains, see § *IV*. Within each committee nodes process transactions and generate sideBlocks, parentBlocks, and state blocks. The primary of the committee collects transactions from transaction pool and includes them into a sideBlock, refer to § *IV-B3*. It then triggers the CoSi-based Zyzzyva consensus algorithm, see § *IV-B1*, to reach agreement among the committee group for the proposed sideBlock. The sideBlock is attached into the sideBlock pool once the committee agreed on it. ZyConChain requires the primary to change in order to bypass the challenges of the view-change phase of the Zyzzyva consensus algorithm, see § *IV-B2*. ZyConChain proceeds based on epochs. In each epoch a leader is elected, which generates parentBlock (which includes a number of sideBlocks) and sate Block, see § *IV-B4*. To address the cross-shard transactions, ZyConChain generates verifiable objects for the cross-shard transactions and includes them into the state block, see § *IV-C*, other shards upon receiving the state block are able to verify the cross-shard transactions confirmation and finalize the cross-shard transaction.

D. ORGANISATION OF THE PAPER

The rest of this article is organized as follows. Section II reviews the main existing solutions, followed by Section III that provides details of the problem to be addressed. Section IV explains in details the proposed ZyConChain protocol. The performance analysis of ZyConChain is presented in Section V, and Section V-C concludes the paper. A roadmap of the paper is also provided at Figure 2.

II. RELATED WORK

This section discusses some of the well-known protocols, outlining both their advantages and limitations.

A. BITCOIN-NG [2]

This system addressed the scalability issue of Bitcoin, and its remarkable finding was that generating block in Bitcoin involves two tasks: (1) solving a hash puzzle (i.e., referred to as Proof-Of-Work (PoW) mechanism, and proving that miner has worked on the block to prevent Sybil attack [24]) and (2) serializing/validating/packing transactions into the block. Based on this finding, Bitcoin-NG broke the Bitcoin's block into two blocks, called *KeyBlock* and *microBlock*, to decouple the two tasks. The former is referred to as leader, and the latter is referred to as transaction proposal. The keyblock includes the solution for the hash puzzle, the PoW, and is generating in every 10 minutes. The miner of the keyBlock becomes then the leader and generates microBlocks (transaction block). The leader generates microBlocks until the next keyBlock is mined and a new leader is elected. Thus, the microBlock generation interval is small, i.e., the microBlocks are generated at high frequency. This obviously increases transaction throughput, however the drawback of this high frequency microBlock generation is the creation of fork on almost every keyblock. Bitcoin-NG is also vulnerable to selfish mining attacks. Furthermore, Bitcoin-NG has not resolved the scalability problem: if the number of miners in a network increases, the network will not be able to scale up.

B. ByzCoin [3]

This system suggested a new structure for blockchain to enhance transaction's throughput and improve performance.

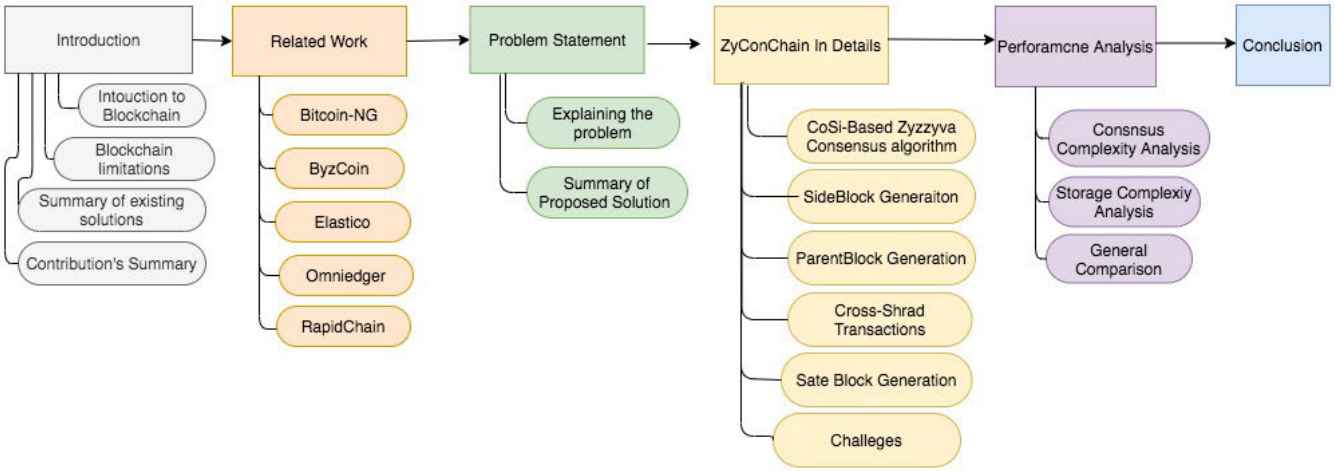

FIGURE 2. Paper road map.

Inspired by Bitcoin-NG [2], ByzCoin decoupled leader election and transaction proposal. Thus, it comprises of KeyBlocks and microBlocks. Similar to Bitcoin-NG, the keyBlock is generated by the PoW mechanism that announces the epoch's leader. The keyBlocks are also used to form a consensus committee by a sliding-share-window mechanism. The consensus committee comprises of the recent keyBlock miners based on the amount of share they own. More specifically, once a miner finds the solution for the hash puzzle and generates the keyBlock, it is credited a share in the current consensus group and moves the share window, which includes the previous committee, one share forward and forms the new consensus group. The miners in the current consensus group can participate in the consensus for the microBlock generation. To generate the microBlocks, ByzCoin behaves different from Bitcoin-NG. It uses PBFT consensus algorithm [19] and collective signature (CoSi) protocol [20] to introduce a scalable PBFT-based consensus algorithm for microBlock generation. Once the new leader is elected and the committee for the current epoch is formed, the leader generates the microBlocks, and triggers the CoSi-based PBFT consensus. Once the consensus is reached on the microBlock, the leader includes the collective signature into the microBlock and adds it to the microBlock chain with a pointer to the preceding KeyBlock. This increases throughput, the issue of the scalability however still remains: when the number of miners in the network increases, ByzCoin can't scale out and increase the transaction throughput. Furthermore, ByzCoin has a flat topology and falls back to this topology in case of failure, which is detected by the keyBlock miner. The flat topology requires all-to-all communication among the committee group. which incurs $O(n^2)$ in the worst case.

C. ELASTICO [4]

This is the first system that is based on a sharding consensus protocol to address the scalability limitation of permissionless blockchains. Elastico partitions a network into smaller committees, where each committee processes a different set of transactions. Each committee has a relatively small number of nodes $\sim$ 100 and runs PBFT consensus algorithm that has $O(n^2)$ communication complexity. Elastico operates in epochs, and re-partitions committees in each epoch. Hence, Elastico's nodes are assigned to a committee only for duration of epoch. At the end of each epoch, nodes solves a hash puzzle (PoW), which is seeded from a random string generated by the final committee from the last epoch; then later send their solution to the final committee to be assigned to a new committee for the next round. Final committee uses the least-significant bits from the PoWs to distribute nodes into committees. After the committee setup, the nodes within the committee agree on a set of transactions and send it to the final committee, which gathers the transactions received from committees into a global block and distributes it to all committees. Elastico does improve throughput and latency of Bitcoin, however it has some limitations: (i) Elastico only partitions the nodes and it does not divide the blockchain. This requires all the nodes to store the entire blockchain. Thus, when a block is generated, it needs to be sent to all the nodes in the network. This incurs a high communication complexity. (ii) The size of each committee is fixed and set to small number $\sim$ 100, and this increases the probability failure to 97% after 6 epochs [5], [6].(iii) Elastico does not provide a solution for Cross-shard transactions [5].

D. OMNILEDGER [5]

This system is also a sharding based protocol that addresses some of the Elastico's limitations. Similar to Elastico, Omniledger is designed for permissionless blockchains. Hence, to allow nodes to join/leave the protocol, it re-configures the committees at every epoch (once a day). The reconfiguration ensures that a committee is never compromised. To achieve this, Omniledger designed a secure committee reconfiguration protocol based

on RandHound [25] and Verifiable Random Function (VRF) [26]. In each epoch, a fresh randomness is generated using RandHound protocol [25]. This randomness is used to partition the nodes into smaller committees. RoundHound however relies on a leader to orchestrate the protocol. Thus, it needs a leader election mechanism that is unpredictable and unbiasable to prevent the adversary to break the protocol. Similar to Algorand [8], Omniledger uses VRF [26] to provide the unbiased and unpredictable leader election mechanism. The consensus protocol within each committee is a variant of ByzCoin [3]. One of the major security weaknesses of ByzCoin's design is the flat topology that the protocol switches into in case of failure, which is a all-to-all communication model in the Byzantine setting. It is not known to whether Omniledger's new scheme of ByzCoin has addressed this issue [6]. Besides, Omniledger has several drawbacks: (i) its cross-sharding mechanism is based on a lock/unlock mechanism which can result in a denial-of-service attack (DoS), as a malicious user can lock arbitrary transactions [6], (ii) Omniledger's consensus protocol requires all nodes in each committee to gossip messages to n nodes for each block, which incurs $O(n)$ per node communication complexity. (iii) Omniledger can have less than 10 seconds low latency if $t < n/8$ [6]. Finally, (iv) Omniledger's cross-sharding transaction mechanism is for *Unspent Transaction Output* data model (UTXO), thus it is restricted to Cryptocurrency applications. Moreover, it is based on a strong assumption (i.e., users must actively participate in the cross-shard transaction), which is not feasible for lightweight nodes.

E. RapidChain [6]

This system aimed to address the drawbacks of previous sharding-based protocols. Unlike previous protocols that perform partial sharding, RapidChain presents the full sharding of computation, storage, and communication overhead of processing transactions. It partitions nodes into smaller groups of nodes, called *committees*, i.e. the computation sharding. Each committee maintains a disjoint ledger (storage sharding) and process disjoint set of blocks. By sharding the storage and the nodes (into committees), RapidChain could scale the system's throughput in proportion to the number of committees. Similar to previous sharding-based protocols, RapidChain operates in epochs. In the first epoch, it configures a committee of size $O(\log n)$ as the reference committee, which is responsible for performing the reconfiguration task between epochs. At the end of every epoch, the reference committee generates a fresh randomness used (i) to divide nodes into committees, (ii) to allow participating nodes to obtain a new identity, and (iii) to reconfigure the existing committees to provide the join/leave property required for permissionless blockchains. For reconfiguration, RapidChain uses Cuckoo rule [27], [28] (a) to provide liveness at the time of reconfiguration (re-organizing only a subset of nodes), (b) to reduce the large communication overheads on the network, and (c) to protect against slowly-adaptive Byzantine adversary to the protocol. Once the committees are reconfigured, nodes receive transactions from the network and start packing them into a block. To reach consensus on a block, RapidChain uses a Synchronous BFT protocol [29] within committees: when the committee agrees on a block, to propagate it within the committee, RapidChain uses a fast gossiping protocol built on the Information dispersal algorithm (IDA) [30], [31]. For communication sharding, RapidChain applied Kademlia routing protocol [32] to divide the communication between committees. Hence, each committee stores a routing table with $\log n$ size of records. When nodes need to communicate with other nodes from other committees, they only take $\log n$ steps to discover them. Despite its advantages, RapidChain has a few limitations: (i) similar to previous sharding protocols, RapidChain is designed for *Unspent Transaction Output* (UTXO) data model, and thus it is limited to Cryptocurrency applications; (ii) it performs cross-sharding transactions by splitting a transaction into three transactions based on the shards they belong to. This approach fails to provide isolation which is one the essential requirements for distributed transactions [33], and (iii), RapidChain design is very complex.

III. PROBLEM STATEMENT AND PROPOSED SOLUTION – A SUMMARY

As detailed in Section II, the current solutions to address scalability have major limitations. ByzCoin & Bitcoin-NG have improved performance, they, however, fail to scale out when the number of miners in the network increases. Recall that scaling out refers to the ability of the system to process more transactions when the number of miners in the network increases. For instance, if a network with N nodes processes T number of transactions per second, when the number of nodes increases to $2N$, then the system should be able to process more transactions than T. Sharding technique [4] is one approach that has been applied in several work, such as Elastico [4], Omniledger [5], and RapidChain [6], to resolve this issue. However, these proposals have some limitations.

There is a major challenge when applying sharding, namely *cross-shard transaction processing*. Cross-shard transaction refers to the transaction that results in the update of two or more shards. For example, assume a transaction $Tx = <(in_1, in_2), O>$, where in_1 is a coin from shard1 and in_2 is a coin from shard2 and these two coins create a new coin, namely O, in shard3. This is a cross-shard transaction. Elastico [4] applied sharding to improve the scalability, but it did not address the cross-shard transaction processing. Omniledger also leverages sharding. For cross-shard transaction, Omniledger, applies a lock/unlock mechanism. RapidChain [6], another solution based on sharding, has applied a different technique to support cross-shard transactions. RapidChain splits the transaction into sub-transactions and process the new sub-transactions separately. In the above example, RapidChain splits the Tx into three transaction, $Tx_1 =< in_1, in'_1 >$, $Tx_2 =< in_2, in'_2 >$, and $Tx_3 =< (in'_1, in'_2), O >$. in'_1, in'_2 belong to shard3. Each of these transactions are processed individually. Here, Tx_1 sends the

in'_1 to shard3, then Tx_2 will send in'_2 to shard3, then in shard3, Tx_3 will combine the in'_1 and in'_2 and create the new coin O. However, the drawback with Omniledger and RapidChain solution is that they are mainly designed for UTXO data model, hence they are limited to Cryptocurrency applications [33]. UTXO (Unspent Transaction Output) is the data structure used in most Cryptocurrency applications. Moreover, RapidChain does not provide isolation which is one the key factor of ACID properties of transaction processing in database systems [33]. It is also a complicated design.

The aim of this article is to design a novel protocol to improve scalability of blockchain that addresses the above limitations and fits different types of applications. We have three goals:

- Increasing throughput
- Reducing latency
- Providing a cross-shard transaction solution for sharding technique that is not limited to a specific data model

Being inspired by ByzCoin and Bitcoin-NG, we separate transaction block and the leader election. However, to improve upon their throughput and latency, we have modified the chain structure, transaction block generation, and the consensus algorithm that is applied in committee group to finalize the transaction blocks. We introduced three types of blocks: *parentBlock*, *sideBlock*, and *state block*. *ParentBlock* is the leader election block and *sideBlock* is the transaction block. ParentBlocks and sideBlocks are maintained in different chains but are linked together. SideBlocks are generated, by nodes in the committee, before parentBlocks are generated, and they are kept in a sideBlock pool. When miners want to generate a parentBlock, instead of packing transactions into the blocks, they pack sideBlocks and attach them into the parentBlock. This increases the sideBlock generation rate, which improves the throughput and latency.

To enable ZyConChain to scale out, we applied sharding technique with the aim not to limit its use to Cryptocurrency applications. We introduced *state block* that contains cross-shard transactions information and forms the *state chain*. Each shard has its own state chain. To facilitate the cross-shard transactions, we designed parallel chains that comprises of state chains of all shards. In another word, we leveraged parallel chains to link different shards together to facilitate cross-shard transactions. This means each shard keeps the state chains of other shards, in parallel, which enables it to verify cross-shard transaction confirmation. Thus, each shard, besides its own chains, has the state chains of all other shards.

In summary, the differences between ZyConChain and existing solutions are as follows:

- In ByzCoin and Bitcoin-NG, once a keyBlock is generated, then the leader will generate microBlocks until the next keyBlock is generated. In ZyConChain however, sideBlocks are generated before the parentBlocks are generated, and they are kept in a sideBlock Pool. To generate a parentBlock, miners create a parentBlock and pick several sideBlocks from pool and attach them into the parentBlock. Then start mining for that parentBlock. Once the parentBlock is mined (i.e., a miner finds the hash for PoW of parentBlock), the parentBlock and its attached sideBlocks are confirmed.
- To finalize the microBlocks, ByzCoin applies the expensive 3-phase protocol PBFT algorithm for each microBlock (i.e., *preprepare*, *prepare*, and *commit*). In ZyConChainn, Zyzzyva consensus algorithm (with a minor modification) is used as a faster consensus algorithm for sideBlock generations.
- The other limitation of ByzCoin and Bitcoin-NG is that, they are not able to scale out. ZyConChain applied sharding technique to address this issue.
- State-of-the art sharding-based protocols (e.g. Elastico, Omniledger, and RapidChain) have a few limitations that ZyConChain attempted to address here.
  - As Elastico is not able to process cross-shard transactions, ZyConChain supports cross-shard transactions.
  - Omniledger applies a lock/unlock mechanism to provide cross-shard transaction. RapidChain divides the cross-shard transactions into separate inter shard transaction to process and finalize them. However, they both are limited to Cryptocurreny applications. ZyConChain provides a new concept, i.e. parallel state chain to enable nodes in different shards to process and verify the cross-shard transaction. This solution is not limited to a specific data model.

IV. ZyConChain IN DETAILS

ZyConChain partitions the nodes into small groups, called *committees*, and enables the network to scale out when the number of nodes increases. Each committee maintains a *main chain, a sideBlock chain*, and *state chain*, depicted in Figure 3. Main chain includes *parentBlocks*; and sideBlock chain comprises of *sideBlocks*. ParentBlock is the block that elects the leader for a committee in each epoch; and sideBlock is the transaction block. State block, as the name suggests, is the state of the committee in epoch. The mined parentBlock and confirmed transactions in the epoch define the state of the committee for that epoch. Hence, the state block includes the hash of the parentBlock generated in that epoch; the merkleRoot of the transactions that are finalized in that epoch; and the list of Cross-shard transactions. The state block is introduced to address cross-shard transaction.

The committee also keeps the state chains of all other committees in the networks, referred to as *parallel chains*. Parallel chains link different shards together. Having the state information of other shards, which includes the cross-shard transaction information, enables the nodes within a shard to verify the cross-shard transaction confirmations. Figure 4 illustrates an example of parallel chain in one shard – **S**hard1.

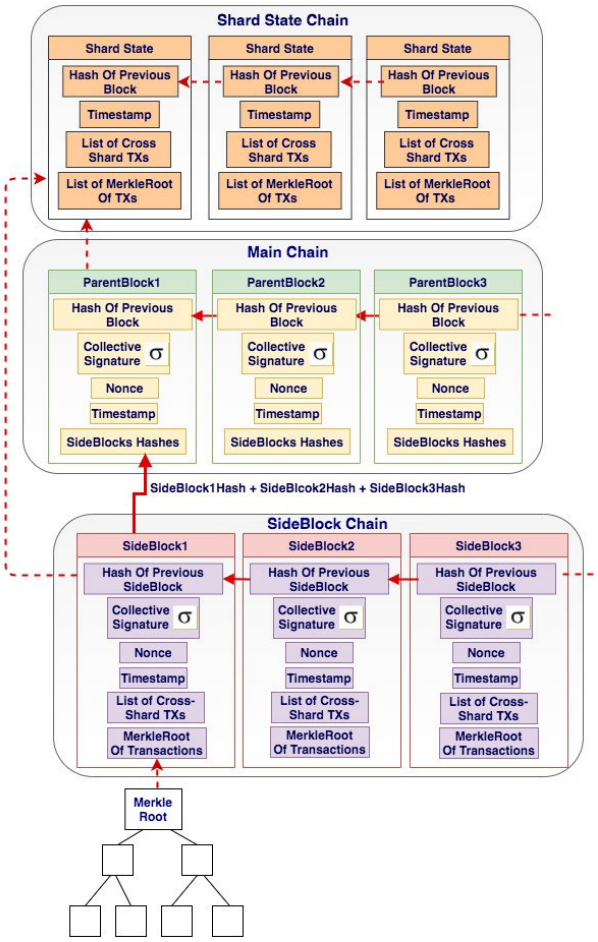

FIGURE 3. ZyConChain blocks structures.

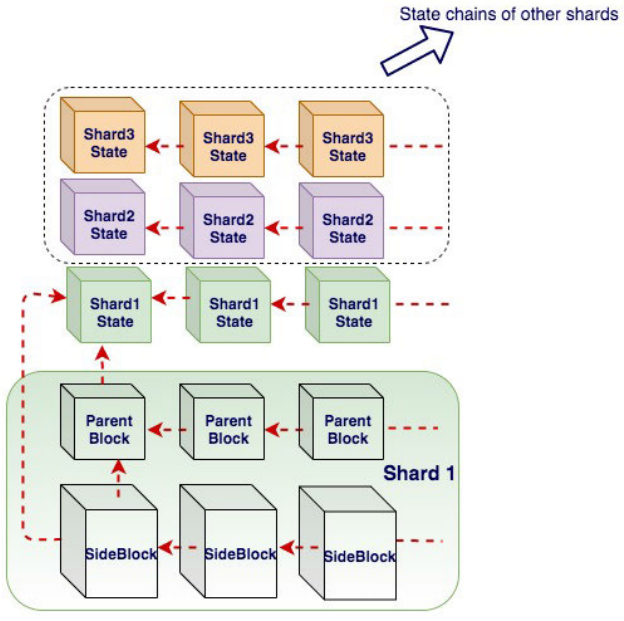

FIGURE 4. Chains within one shard.

A. NETWORK MODEL

We assume a network with n nodes over a peer-to-peer partially synchronous setting. In a partially synchronous model, there is no bounded time Δ for message delivery. This means, messages are guaranteed to be delivered but there is no known delay time for delivery [34] (for more details in network synchrony refer to Appendix A). We also assume all nodes in the network have equivalent computational resources. Each node has a private-public key set, (pk_i, sk_i), as an identity. The nodes are divided into $c = n/m$ shards, where $m = t \log n$ is the size of each shards, refer to as committee.

B. ZyConChain WITHIN ONE COMMITTEE

In ZyConChain blocks are generated within two different layers, namely *Committee-layer* and *Leader-layer*.

- Committee-layer: SideBlocks are generated in this layer. In each committee, nodes first create SideBlocks, and later run the consensus algorithm to get other nodes agreement on the sideBlocks. Once other nodes agreed on the sideBlock, sideBlock is included into a sideBlock pool, where they are waiting for miners to collect them and attach them into a parentBlock. While they are in the pool, they are not finalized yet. But, once they are attached to a parentBlock and the parentBlock is added into the main chain, then the sideBlocks are finalized. Because all nodes participate in generating sideBlocks and consensus algorithm, we refer to the sideBlock generation layer as the committee-layer.
- Leader-layer: ParentBlocks and State blocks are generated in this layer. To generate a parentBlock, nodes first create a parentBlock and pick several sideBlocks from pool and attach them into the parentBlock. They, then, start mining for that parentBlock. Once the parentBlock is mined (i.e., a miner finds the hash for the parentBlock, PoW mechanism), the parentBlock and its attached sideBlocks are finalized. The miner which has created the current parentBlock will become the leader of the epoch. The leader will then create the state block of the epoch and collect the committee group's signatures for the state block. Finally, the state block is included in the state chain and the leader will broadcast it to other shards in the network, using Kademlia routing protocol [32].

1) CoSi-BASED Zyzzyva CONSENSUS PROTOCOL

To reach consensus for a sideBlock in a committee, we designed a novel consensus algorithm based on Zyzzyva, i.e., a BFT consensus algorithm in distributed systems. To enable Zyzzyva to scale for a large network, we applied CoSi protocol [20], which is a scalable collective signature protocol based on Schnorr signature. This section provides details of *CoSi-based Zyzzyva*.

Let us consider a committee of size $N = 3f + 1$, where N is the total number of nodes in the committee and f is the number of Byzantine nodes. CoSi-Based Zyzzyva Consensus comprises of two paths, namely *fast path* and *two-phase path*, which resembles to the PBFT protocol. In the normal condition (i.e. all nodes are correct nodes), the *fast path* will

be executed and terminate the consensus. However, if the fast path fails to complete, the protocol falls into the two-phase path. The paths are explained as follows:

- *Fast path*
  - First, the primary (explained in § *IV-B2*), computes a hash for a sideBlock and sends the generated sideBlock to other committee members.
  - Second, the committee members will validate the transactions in the sideBlocks and add their signature if the transactions and the hash value are valid.
  - Third, the primary will collect all the signatures from the committee members using CoSi protocol [20]. If $3f+1$ signatures are collected (including the primary's signature), the consensus has reached for the sideBlock and it is confirmed. This path is complete here.
- *Two-phase path*
  If the primary does not receive $3f+1$ signatures for the proposed sideBlock, then the protocol falls in this path. In this path, assuming that the primary receives between $2f+1 \leq Signatures < 3f$, then it will create a *commit certificate* and send it to other committee members along with the $2f+1$ signatures collected from previous path. If the primary receives back $2f+1$ signatures acknowledging the *commit* message, then the primary will add the sideBlock into its pool and send the *commit* message to other committee members along with the collected signature from $2f+1$ nodes. This path is complete here.

2) VIEW CHANGE

Zyzzyva is known to have a complex view change mechanism. In leader-based-BFT consensus algorithms the view change occurs when the primary (leader) is faulty. This is because the primary is fixed. Thus, when the nodes detect that the primary is faulty or behaving maliciously, they trigger the view change to change the primary. We simplified the Zyzzyva's view change mechanism by avoiding to have a fixed primary. That is the primary is changing in each epoch based on a PoW mechanism (with a low difficulty such that in every 5 minutes a new primary is elected).

To change the view in ZyConChain nodes work to find a solution for PoW puzzle (i.e., finding a hash on a set of data (*timestamp, nonce, previous primary's hash*) that is less than a target). Upon finding the hash the node, i.e., the *new primary*, sends the hash, a *primary commit* message, the set of data, and the last sideBlock's id it has received from the current primary, to all the nodes in the committee. The nodes upon receiving the new primary's hash, (i) verify the correctness of the hash, and (ii) check whether the last sideBlock's id that the primary has sent is similar to the last one they have in their pool. If the conditions are met, nodes (1) append the new primary's hash into their *view list* (fig 5), (2) stop receiving the sideBlocks from the current primary, and (3) send their signature on the *primary commit* message to the new primary. Upon receiving $2f+1$ signatures the new primary sends the collected signature to nodes and start generating sideBlocks until the next primary is elected. Note that the communication overhead for the view change is $\log n$ as the new primary uses CoSi protocol [20] to collect the signature for *primary commit* message.

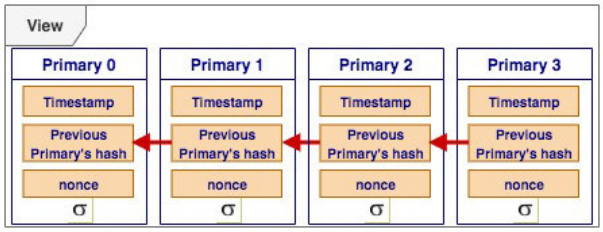

FIGURE 5. View list.

3) HOW SideBlocks ARE GENERATED IN EACH COMMITTEE?

To generate a sideBlock, nodes follow the below steps in each committee:

- The primary creates a sideBlock, includes transactions into it and computes a hash for the sideBlock.
- Then the primary, needs to collect signatures of the committee group for the newly generated sideBlock. Hence, it uses the CoSi-based Zyzzyva consensus protocol, to collect $3f+1$ signatures.
- If $3f+1$ signatures are collected (including the priamry's signature) the sideBlock has been accepted by the committee. The primary will include the collective signature, σ, into the sideBlock and add it to the pool, where it is chained to the previous sideBlock and waits to be picked. It, then, sends the σ to other nodes so that they can include it into the sideBlock and add the sideBlock to their pool. Here, the consensus is terminated.
- But, if $2f+1 \leq number\ of\ Signatures < 3f$, then the protocol will fall into the second phase of the CoSi-Based Zyzzyva protocol, where the primary sends a *commit certificate* along with the collected signatures to the nodes in the committee. Then the primary waits to collect another $2f+1$ signature for the *commit certificate*. If it receives the $2f+1$ signatures, then the sideBlock is considered accepted and can be added to the pool. Then primary, will send the sideBlock along the *commit certificate* and the collected signature to the nodes, so nodes add the sideBlock into their pool.
- if less than $2f+1$ signatures are collected, then the sideBlock is failed and not accepted.

When collecting signatures in the Zyzzyva consensus rounds using CoSi protocol [20], we reduced the communication cost to $\log n$, where n is the total number of nodes in the committee. To verify σ, the final signature, nodes perform $O(1)$ computation. Hence, the total computation and communication costs for generating each sideBlock is $O(\log n) + O(1) = O(\log n)$. SideBlocks are generating with high rate, every few seconds. We then pool the generated sideBlocks,

Algorithm 1 SideBlock Generation Algorithm

Input: *List of transaction*
Output: *sideBlock*

```
Function sideBlockGen():
    SideBlock sideBlock ← new SideBlock();
    List<Transactions> TXs ← List of Transactions;
    sideBlock.add(TXs);
    merkleRootTXs ← MerkleTree().getMerkleRoot(TXs);
    List<CrossShardTransaction> crossTXs ← List of Cross-Shard Transactions;
    sideBHeader ← previousSideBHash + timestamp +merkleRootTXs + crossTXs;
    while !sideBHash < target do
        nonce + +; // difficulty is low to find hash every few seconds
        sideBHash ← sha256(sideBHeader|nonce);

    end while
    σ ← CoSiBasedZyzzyvaConsensus(sideBlock);
    sideBlock.addCollectiveSignature(σ);
    return newSideBlock;
End Function // Run the consensus algorithm within the committee.
Function CoSiBasedZyzzyvaConsensus(sideBlock):
    σ ← collect committees' signatures;
    return σ;
End Function
```

and this is introduced to facilitate the sideBlock generation in parallel with parentBlock generation (see Algorithm 1).

4) HOW ParentBlocks ARE GENERATED?

In each committee, nodes, parallel to generating sideBlocks, participate in PoW mechanism to find the hash for PoW puzzle and generate a ParentBlock. The steps to generate a parentBlock are as follows:

- First, node creates a parentBlock, pick several sideBlocks from sideBlock Pool and attach them to the parentBlock (i.e., adding the hash of the sideBlocks to the parentBlock). Then it starts mining for the parentBlock. Mining is based on the PoW mechanism.
- The node which finds the hash, which means the parentBlock is mined, will become the leader and add the parentBlock to main chain and also add the sideBlocks into the sideBlock chain. Next, it broadcasts the newly mined parentBlock to all nodes within the committee. Nodes in the committee, following Nakamoto's consensus algorithm (i.e., the longest chain rule), validate the parentBlock then either accept it or reject it. If nodes accept the parentBlock, they will add it to their copy of main chain and add the sideBlocks, which are attached to the parentBlock into the sideBlock chain. Note that only parentBlock is sent to other nodes. Because nodes within the committee already have the sideBlocks in their sideBlock pool, they hence are able to retrieve them and add them to the sideBlock chain.
- The leader, which mined the current parentBlock, then creates a state block for this epoch. It includes the following information into the state blocks:
  - MerkleRoot of all the transactions finlaized in this epoch. The merkleRoot is available in the sideBlock. Hence, depending on the number of sideBlocks that leader has attached to the parentBlock, the number of MerkleRoots included into the state block varies. For instance, if the leader has attached 3 sideBlocks to the parentBlock, then it needs to include 3 merkleRoot to the state Block.
  - List of cross-shard transactions. It is available in the sideBlocks. Similar to the merkleRoot, the number of cross-shard transactions depends on the number of sideBlocks attached to the parentBlock.
  - The hash of the current parentBlock. This is used as a pointer to point to the previous state block in the state Block chain to form the chain.
  - Then leader requires to collect the committee signatures for the generated state block. Hence, it triggers the CoSi Zyzzyva consensus algorithm to collect the signatures.
  - Once the signatures are collected, then the state block is finalized and can be appended to the state chain.
- The leader's last task in the epoch is to broadcast the state block to other shards in the network.

Algorithms 2,3 are the parentBlock and state block generation algorithms. Figure 6 shows an overview of the sideBlock and parentBlock generation in one committee. Figure 7 depicts the state chains update in two different shards. Algorithm 4 is the ZyConChain main algorithm.

C. CROSS-SHARD TRANSACTION AND PARALLEL CHAINS

As explained earlier, each committee has 3 chains: *main chain*, *sideBlock chain*, and *state chain*. ZyConChain also introduces parallel chains structure (i.e., state chains of all shards in the network) to link shards together. To do so, state chain of each shard is kept in the other shard. Therefore, to link all the shards together, each shard keeps the state chain of all other shards, i.e., parallel chains structure. This is done to facilitate cross-shard transactions. The state chain comprises of state blocks generated in each epoch. Each state block contains, hash of the parentBlock generated in that epoch, the merkleRoot of the transactions that are finalized in that epoch, and the list of Cross Shard Transactions, i.e. shown as $Cross\_Tx$. Having the merkleRoot of transactions and the cross-shard transactions information, which are stored in the state blocks, enables the nodes within the shard to verify whether a cross-shard transaction has been confirmed.

Algorithm 2 ParentBlock Generation Algorithm

Input: *List of SideBlocks*
Output: *parentBlock*

```
Function parentBlockGen():
    ParentBlock parentBlock ← new ParentBlock();
    List<SideBlock> sideBlocks ← List of SideBlocks;
    parentBlock.attachSideBlocks(sideBlocks);
    List<Hash> sideBHashes;
    foreach SideBlock si : sideBlocks do
        sideBHashes.add(si.getHash());
    end foreach
    parentBHeader ← previousparentBHash +
    timestamp + sideBHashes
    while !parentBlockHash < target do
        // PoW mining.
        nonce + +;
        parentBHash ←
        sha256(parentBHeader|nonce);
    end while
    return newParentBlock;
End Function
```

Algorithm 3 Shard State Block Generation Algorithm

Input: *parentBlockHash, List of Cross-Shard Transactions, List of MerkleRoot of Transactions*
Output: *stateBlock*

```
Function shardStateBlockGen():
    StateBlock stateBlock ← StateBlock();
    List<CrossShardTransaction> crossTXs ← List of
    Cross-Shard Transactions;
    List<MerkleRoot> merkleRootTXs ← List of
    MerkleRoots of Transactions;
    foreach SideBlock si : sideBlocks do
        crossTXs.add(si.getCrossTXs());
        merkleRootTXs.add(si.getMerkleRootTXs());
    end foreach
    stateBlock ← parentBHash + timestamp
    +crossTXs + merkleRootTXs;
    stateBlock.addHash.(sha256(parentBHash|
    timestamp |crossTXs|merkleRootTXs));
    return newStateBlock;
End Function
```

Let us assume we have a transaction $Tx = <(acc1, acc2), amnt>$, where $acc1$ is an account in shard1, $acc2$ is an account which belongs to shard2, and $amnt$ is the amount to transfer from $acc1$ to $acc2$. Also assume this transaction is sent to the shard1. To process this transaction, our protocol follows the steps below:

- The transaction is divided into two transactions, $Tx = Tx\_1 + Tx\_2$. Tx_1 is to deduct the amount from $acc1$. $Tx\_1 = < acc1, -amnt >$. And Tx_2 is to add the amount to $acc2$. $Tx\_2 = < acc2, +amnt >$.
- An object $Cross\_Tx$ is created to contains the followings: the original transaction id (Tx_Id), the split transactions ids, (Tx_1_Id and $Tx\_2 - Id$), two verifiable Objects ($VO1$ and $VO2$) for Tx_1 and Tx_2 which are initially set to null. VO is the path of a transaction in the merkle Tree. This path enables a verifier to verify whether a transaction has been processed by reconstructing the merkleRoot. Note that, to reconstruct a merkleRoot, the transaction Id and the path of that transaction in the merkleTree is sufficient. If the reconstructed merkleRoot is the same as the merkleRoot stored in the state block, then the transaction has been confirmed and everyone can verify that. Figure 8, depicts the merkle Tree and the path to verify a transaction in the merkle Tree.
- Tx_1 and the $Cross\_Tx$ will be sent to shard1.
- Shard1 then processes the Tx_1 and updates the $Cross\_Tx$ by adding the merkle tree path of the Tx_1 into the $VO1$. Once the state block is generated and broadcast, other shards are able to verify whether the Tx_1 is confirmed.
- Then the Tx_2 along with the $Cross\_Tx$ is sent to shard2.
- Shard2 process the Tx_2 and updates the $Cross\_Tx$. Once the Tx_2 is confirmed and the state block is broadcast, all shards can verify the Tx_2.
- If shard2 does not process and confirm the Tx_2 within $\triangle$, then a new transaction Tx_revert is generated, i.e., $Tx\_revert =< acc1, +amnt >$ which must be sent to shard1, but, to avoid double spending attack, i.e., to make sure that the shard2 will not process the Tx_2 sometime in future, we create another transaction $Tx2\_abort$, which is sent to shard2. Once, the confirmation of Tx_abort is received (via state block) then the Tx_revert is sent to shard1.

We assume that the number of cross-shard transactions that are generated between two shards are limited. If the number of cross-shard transactions between two shards, say shard1 and shard2 is more than a threshold, then it means that the nodes are not partitioned correctly. In this case, the network require re-partitioning.

D. CHALLENGES

We faced the following challenges when designing ZyConChain.

- Preventing Sybil attacks. To prevent this attack, we have applied PoW mechanism, which its security has been proven and it has been used in many blockchain protocols

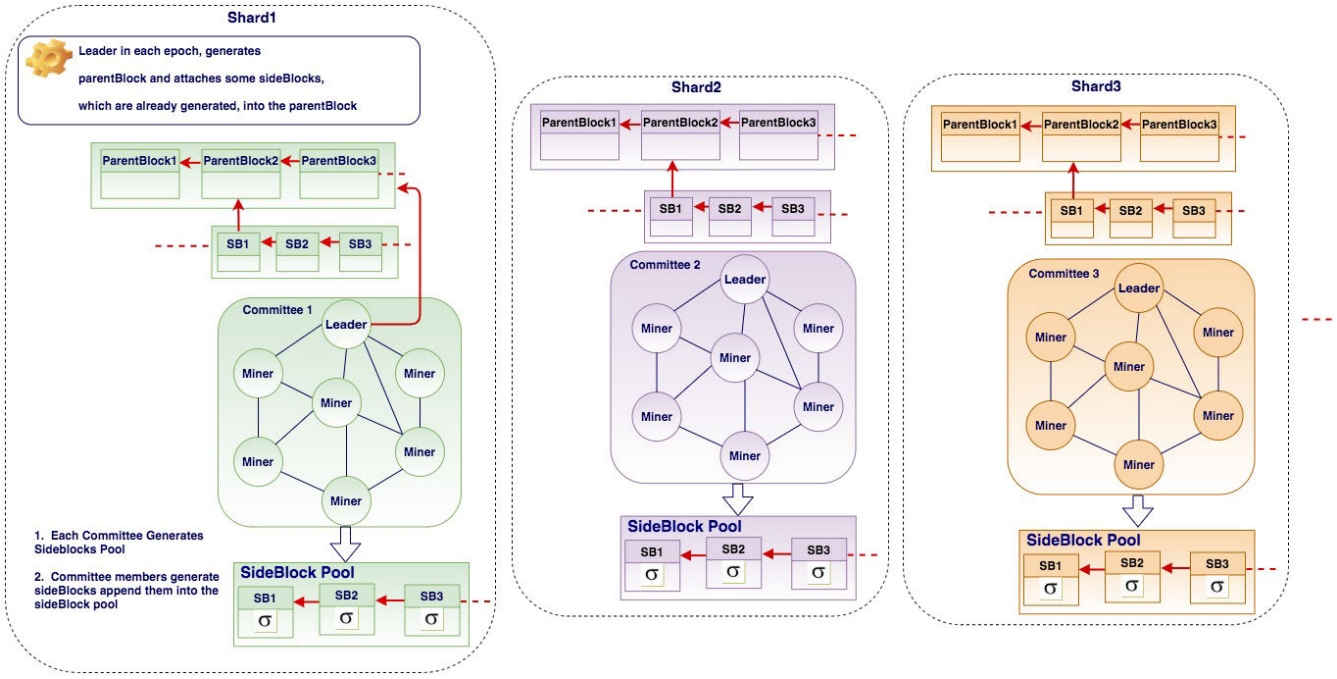

FIGURE 6. Within committee ParentBlock and SideBlock generation overview.

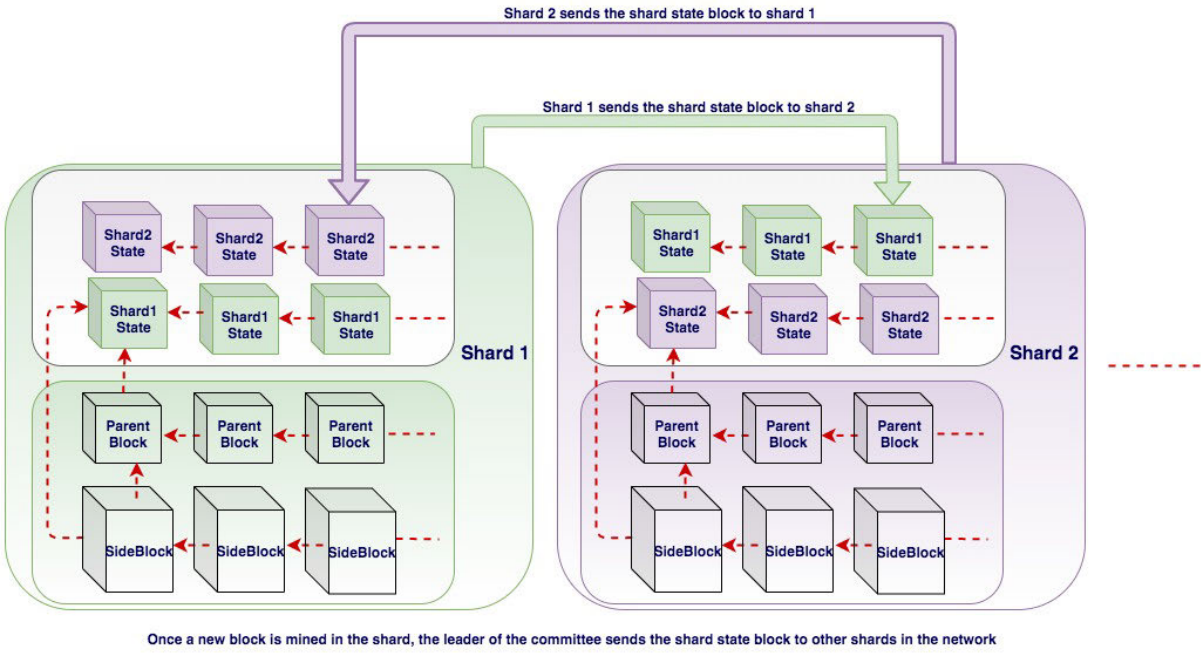

FIGURE 7. Parallel chains in two shards.

- Securing the sideBlocks. We first need to define what the security of a sideBlock means. As mentioned, every node in each committee has a sideBlock pool along with the main chain. We designed sideBlock pool to enable fast generation of sideBlocks. However, once a sideBlock is appended to a pool, no one should be able to modify it. Furthermore, the total order of transactions must be kept. These are the security requirements of a sideBlock. To provide immutability and total transaction ordering, every sideBlock has a hash and also is linked

Algorithm 4 Main Algorithm

```
// joining network and mining.
Join the network by connecting to known peers;
Start sideBlockGen();
Start parentBlockGen();
// Main loop.
while alive do
    if parentBlockGen() returns parentBlock then
        // add the parentBlock into
           main chain.
        mainChain.add(parentBlock);
        broadcast(parentBlock); // to peers in
           the committee
        List<SideBlock> sideBlocks =
           parentBlock.getSideBlocks(); // get
           attached sideBlocks.
        sideBlockChain.add(sideBlocks);
        sideBlockPool.remove(sideBlocks);
        stateBlock ← shardStateBlockGen();
        broadcast(stateBlock); // to all
           committees.
        resetParentBlockGen();
    end if
    if parentBlock receives & isValid() &
    extendsTheLongestMainChain() then
        mainChain.add(parentBlock);
        List<SideBlock> sideBlocks =
        parentBlock.getSideBlocks();
        sideBlockChain.add(sideBlocks);
        sideBlockPool.remove(sideBlocks);
        gossip(parentBlock);
        resetParentBlockGen();
    end if
    if primary then
        while new primary not elected do
            sideBlock ← sideBlockGen()
            sideBlockPool.add(sideBlock);
            broadcast(sideBlock);// to peers
               in the committee.
            resetSideBlockGen();

        end while
    else if sideBlock receives & isValid()
    sideBlockPool.add(sideBlock);

end while
Function sideBlockGen():
    return sideBlock // Algorithm 1
End Function
Function parentBlockGen():
    return parentBlock // Algorithm 2
End Function
Function shardStateBlockGen():
    return stateBlock // Algorithm 3
End Function
```

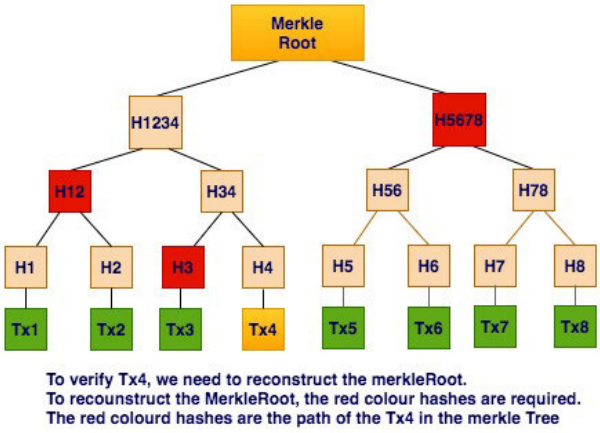

FIGURE 8. Merkle root.

to the previous sideBlock by containing the previous sideBlock's hash. The hash is generated with standard hash functions, namely, SHA-256.

- Double-spending attacks. In ZyConChain, if adversary wants to double-spend on a transaction that is included in t sideBlock before, then it requires to re-generate all the subsequent sideBlocks, which then requires to collect the signatures of the committee. In addition, it requires to re-generate all the subsequent parentBlocks, which to do that, it requires to gain 50% of the hashing power of the entire network. This suggests that, as long as adversary has less than 50% of the power of the network, and also as long as $2/3$ of the nodes in each committee are honest, then the proposed protocol is secure against this attack.
- Scaling the consensus in each committee. similar to ByzCoin, we applied CoSi protocol [20] to scale the Zyzzyva protocol that we have applied in each committee.
- Leader election in each committee. In each committee, a leader is elected based on PoW consensus. The difficulty level of PoW is set to 10 minutes to prevent Sybil attack.

V. PERFORMANCE ANALYSIS

This section analyses the performance of ZyConChian and benchmarks it with some existing work. The following two metrics are considered here:

- Consensus complexity aims to measure the communication and computation costs of ZyConChain's transaction. This relates to the overheads of the modifications of the consensus algorithm (CoSi-Based Zyzzyva Consensus), the block generation, and the chain structures. We then compare the complexity of ZyConChain with the existing work to evaluate its efficiency.
- Storage complexity is used to measure the storage overhead that parallel state chains structure, which every committee needs to store, incurred to the protocol.

A. CONSENSUS COMPLEXITY

We compute the complexity of the consensus per transaction, denoted as Tx. Let us assume that Tx belongs to committee C_i, which includes m nodes. The network is partitioned into n/m committees, where n is the total number of nodes in the network and m is the size of each committee, i.e. $m = t \log n$, and t is the security parameter. Kademlia [32] routing protocol is used for the committee-committee communication. Recall that in Kademlia, each node stores a routing table of size $\log n$, where n is the size of entire network. To discover the nodes and rout messages, Kademlia requires $\log n$ steps. Hence, in the proposed solution, each node can send a message to another node in another committee in $\log n/m$ steps, as each committee requires to store a table of size $\log n/m$ records. Thus, when Tx is first generated, it is sent to a constant number of nodes in the network, which then is sent to committee C_i. To discover the nodes in another committee, the communication cost is $\log n/m$. Thus, to send the Tx to all nodes in C_i, the communication and computation overhead is $m \log(n/m) = O(m \log n)$. Second, the nodes in C_i add Tx into sideBlock and run the consensus algorithm, and has communication overhead of the order of $O(\log m)$. This is because we applied the scalable collective signature (CoSi protocol [20]), and the communication costs have been reduced from $O(m)$ to $O(\log m)$. Next, sideBlock will be attached into a parentBlock and the consensus algorithm is triggered on parentBlock. This adds $O(m)$ communication overhead. Hence, the total communication and complexity of the consensus per-transaction is in the order of $O(m \log n) + O(\log m) + O(m) = O(m \log n + \log(m) + m)$.

TABLE 1. Complexity comparison.

Protocol	Consensus Complexity	Storage Complexity
Elastico [4]	$O(m^2/b + n)$	$O(\|B\|)$
Omniledger [5]	$\Omega(m^2/b + n)$	$O(\|B\|/C)$
RapidChain [6]	$O(m^2/b + m \log n)$	$O(\|B\|/C)$
ZyConChain (ours)	$O((m \log n) + \log(m)/ + m)$	$O(\|B\|/C + C \times \|S\|)$

Table 1 provides a summary of the complexity evaluation of the proposed protocol and existing sharding-based protocols. The consensus complexity comparison is also depicted in Figure 9, which shows that the proposed protocol outperforms existing ones when the number of nodes in the network increases. Note that the committee-to-committee communication cost of the proposed protocol and RapidChain is similar, i.e. $O(m \log n)$, as they both use the Kademlia protocol. RapidChain has however a cost in the order of $O(m^2/b)$, as it has used inter-committee consensus that was different from the one used in the proposed protocol. As we have added an extra step, which attaches the sideBlocks to a parentBlock, this adds an extra cost in the order of $O(m)$. In total however, the proposed protocol still performs better in terms of the consensus complexity.

B. STORAGE COMPLEXITY

Let denotes by $|B|$ the size of Blockchain, which in the proposed protocol is computed by $|B| = |PB| + |SB|$, where |PB| denoting main chain (ParentBlocks) size and |SB| denoting SideBlock chain size, respectively. We divide the blockchain into C shards, where $C = n/m$. Hence, each committee stores $O(|B|/C) = O(m \times |B|/n)$ parts of blockchain; and thus each node stores $O(|B|/C)$ amount of data. We also store the *shard state chain*. Let $|S|$ denotes shard state chain size. As the size of each state block is very small, hence the storage size required to store a state chain is equivalent to the storage size that a light-node needs to store the block headers in conventional blockchains. As mentioned earlier, each committee stores the state chains of other shards, thus each node stores $C \times |S|$ amount of data for state chains. Note that the size of each state chain, denoted as $|S|$, in comparison with the size of total blockchain $|B|$ is negligible. However, as we store the state chains of all shards, it adds some storage costs, which were included here to make the computation much more precise. Therefore, the total storage size that each node requires is $O(|B|/C + C \times |S|)$. As shown in Table 1, the storage complexity of the proposed protocol in comparison with Elastico is significantly reduced. Compared however to Omniledger and RapidChain, the proposed protocol requires slightly more storage, which is due to the parallel chain design.

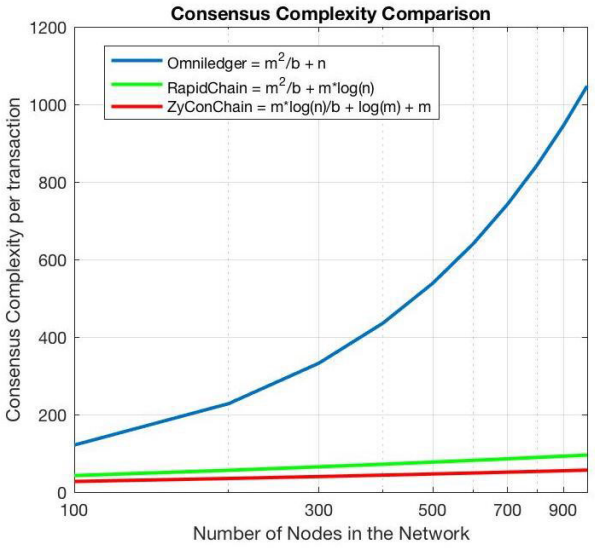

FIGURE 9. Comparison of consensus complexity.

To better understand the storage complexity comparison between the protocols, we plotted the functions from Table 1, in Matlab based on an example explained below.

Let us assume that the total size of the blockchain is $|B| = 200$ MB. We then consider a network with $C = 10$ shards. We also set the size of a state block to 800 B (as mentioned, the size of a state block is small, in reality this size could be even smaller than this, e.g., 80B similar to block header in Bitcoin). Assume each shard has generated 100 blocks, hence, $|S| = 100 \times 800B$. Finally, the graphs are plotted based on these data. As depicted in Figure 10, the difference between the amount of storage required per each node in ZyConChain and Elastico is significant, where

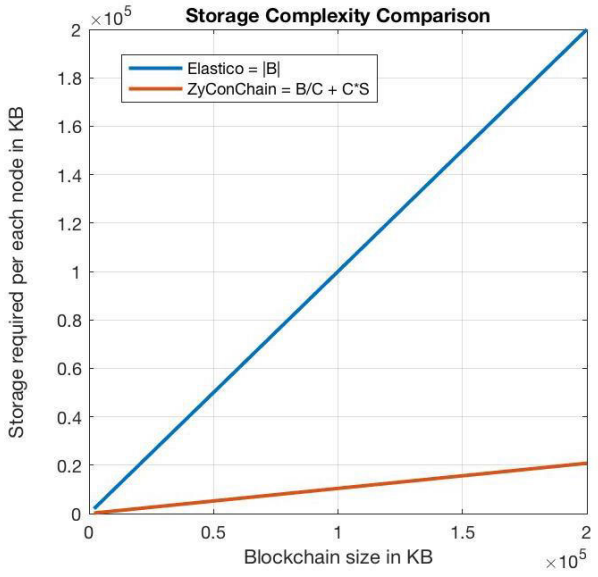

FIGURE 10. Comparison of storage complexity - ZyConChain(Ours) and Elastico.

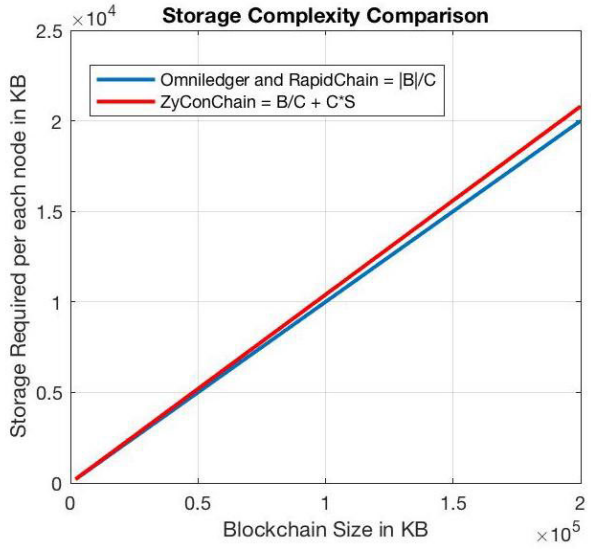

FIGURE 11. Comparison of storage complexity - ZyConChain(ours) and Omniledger and RapidChain.

Elastico requires much more storage space per each node. In comparison with Omniledger and RapidChain however, ZyConChain requires slightly more storage per node, as illustrated in Figure 11.

C. CONCLUDING REMARKS

To the best of our knowledge, this work is the first to propose the concept of the parallel state chains to facilitate the cross-shard transactions for general data models. Table 2 compared the proposed protocol with the exiting sharding-based and some of the standard blockchain protocols. This table shows the advantages of ZyConChain as a scalable blockchain protocol for general applications. ZyConChain provided three types of blocks that form three separate chains. To improve transaction scalability, ZyConChain packs transactions in blocks with a high rate, using Zyzzyva consensus algorithm, and maintains them in a pool. Miners then pick the pooled blocks and attach them into a parentBlock to be finalized. This has significantly enhanced transaction's throughput and latency. Furthermore, ZyConChain uses sharding to scale out when the number of nodes in the system increases. The key idea of ZyConChain is the parallel state chain structure which has been used to facilitate the cross-sharding transactions for any type of data model.

TABLE 2. Comparison of Blockchain protocols.

	Protocol	Scale-Out	Cross-Shard Transaction	Shard State chain	Transaction Model
Non-Sharding Protocols	Nakamoto	×	N/A	×	UTXO
	Bitcoin-NG	×	N/A	×	UTXO
	ByzCoin	×	N/A	×	UTXO
Sharding-Based Protocols	Elastico	✓	×	×	UTXO
	Omniledger	✓	✓	×	UTXO
	RapidChain	✓	✓	×	UTXO
	ZyConChain (ours)	✓	✓	✓	General Data Model

APPENDICES

APPENDIX A–DISTRIBUTED SYSTEMS AND CONSENSUS PROTOCOLS

1) DISTRIBUTED SYSTEM OVERVIEW

A distributed system is a system that its components (hardware or software) are located on the different networked computers and communicate and coordinate their actions by passing messages to one another [34]–[36]. Based on their message exchanging mechanism, distributed systems can be categorised into two types: *message passing* and *shared memory* [16], [37]:

- Shared memory – all nodes (networked components) in the network has access to a shared memory to exchange information to one another.
- Message passing – each node in the system has its own private memory (distribute memory). Information between nodes are exchanged by passing messages.

There are several basic architectures for implementing the applications (called distributed programming) running on distributed systems namely, *client-server*, *three-tier*, *n-tier*, and *peer-to-peer* models [38].

- Client-server – in this model, each node in the system take a role of being server or client. The client interacts with server, which potentially is located in separate host computer, in order to access a shared resource [35].
- Three-tier – the three-tier architecture divides the client, in the client-server model, into two nodes by separating the end-user view from the application logic. Hence, it adds one extra tier to the client-server model. To demonstrate this model, we divide the functions of a given application into *presentation*, *application*, and *data* logic. This application is an example of three-tier architecture [35].

- n-tier – the approach applied in the three-tier can be generalized to n-tiered model. This architecture is a solution for an application which its domain is divided into n logical components and each component is mapped to a separate server.
- Peer-to-peer – in this architecture, all nodes (known as peers) have the same responsibilities and are not separated by their role (client or server). Peers can serve both as clients and as servers. The main goal of the peer-to-peer architecture is to share the data and resources on a very large scale by removing the requirement for a server to manage them. Examples of this architecture includes the bitcoin network.

In this research our focus is on the *message passing* type of distributed systems built on peer-to-peer architecture, as this type of system resembles blockchains.

2) CONSENSUS PROBLEM

Distributed systems are associated with a fundamental problem, i.e., to achieve reliability in the presence of a number of faulty processes (nodes) in the system [16], [34], [39]. This problem, referred to as *consensus problem*. Nodes require to agree on a data value (e.g., a block of data), however, some nodes may be faulty or unreliable. We study two types of nodes faults (or failure): *Crash failure* and *Byzantine failure*.

- Crash failure – it occurs when a node stops its activity, abruptly, and does not resume its functions. In this failure, other nodes in the system can detect the crash [16], [34], [39].
- Byzantine failure – in this failure the node behaves arbitrarily and no assumption can be made about its behaviour. It may send conflicting messages to other nodes or it may remain silent and act as dead for a while then revive itself [16], [34], [35], [39], [40]. Byzantine problem was first introduced by Lamport *et al.* [40], in *Byzantine Generals problem*. In Byzantine generals problem, there are three or more generals that have surrounded a city and need to agree to either attack to the city or retreat. One of the generals, the commander, orders. Others must decide whether attack or retreat abased on the commander order. If the commander is treacherous, he will order attack to one of the general, and retreat to the other.

Thus, to reach agreement among nodes in distributed systems, *consensus protocols* are required and they need to tolerate the aforementioned failures.

A consensus protocol runs among the nodes with the goal of reaching agreement over a data value [34], [35]. A consensus protocol is *Crash Fault Tolerant* (CFT) if it can tolerate a number of crash failures, or *Byzantine Fault Tolerant* (BFT) if it can tolerate Byzantine failures [16], [34].

Besides the above failures that consensus protocols need to tolerate, there are two properties that consensus algorithms must satisfy: *safety* and *liveness*. According to Alpern and Schneider [41] *safety* and *liveness* are defined as follows:

- *safety* – all nodes must execute the requests in the same order.
- *liveness* – no valid request is ignored. In another word, all valid requests must be served.

3) BFT CONSENSUS PROTOCOLS

In 1980, Pease *et al.* proved that to design a BFT consensus protocol that can tolerate f faulty nodes, $3f + 1$ total number of nodes is required. This was the foundation of the subsequent BFT consensus protocols. There are also a number of requirements that BFT consensus must satisfy [16], [34], [35], [42]–[45]:

- Agreement – if a correct node decides on an output y'(e.g., a block), then all correct nodes must decide on the same y'.
- Integrity – the decision and the output y' of every correct node must have been proposed by some correct node. That is, the origin of the decision is important and is not from an adversary [34].
- Termination – all correct nodes eventually decides an output.
- validity – if all correct nodes broadcast a valid request m, then m will be eventually delivered and included in an output y'.

To evaluate the performance of consensus protocols often two metrics are considered: *running time* and *message complexity*.

- Running time – refers to the number of rounds that messages need to exchange to reach agreement.
- Message complexity – message traffic generated through running the protocol.

Hence, throughout this research when we need to evaluate the performance of the consensus protocols, we measure the above metrics.

4) NETWORK SYNCHRONY

Network synchrony is a fundamental concept in the distributed systems and refers to the model based on which the nodes (networked components) are coordinating together. It is classified into three types, namely *synchronous, partially synchronous*, and *asynchronous* [16], [17], [34], [42], [43]:

- Synchronous – synchronous network proceeds in a sequence of rounds. In another word, nodes' operations are coordinated in rounds. In each round all nodes perform the operations of that round (e.g., sending a message), and, then move to the next round. The clock drift ratio (which refers to the time at which a computer clock deviates from a reference clock [35]) which can affect the coordination, is addressed by a central clock synchronization service [34]. In this model messages, that are transmitted to the nodes, are delivered within a known bounded time Δ.
- Partially synchronous – in this model, there is no bounded time Δ for message delivery. This means, messages are guaranteed to be delivered but there is no

known delay time for delivery. This causes the nodes to coordinate loosely [34].

- Asynchronous – in asynchronous network, there is no guarantee for message delivery and messages are delivered arbitrarily. That, one message may get delivered instantly, and other may take several years to get delivered. Also, nodes operations are not coordinated, due to the lack of clock synchronization. One node may execute one step of the operations within a short time, while other may take several days to execute that [35].

Partially synchronous network condition is assumed in most practical distributed systems. Hence, we focus on the consensus protocols for partially synchronous network model.

When designing/evaluating consensus protocols this important factor, network model, needs to be taken into account.

5) CONSENSUS PROTOCOLS FOR PARTIALLY SYNCHRONOUS

There are several consensus protocol designed for this network model, namely, DLS-protocol [43], Paxos [46], viewstampeld replication (VR) [47], Practical Byzantine Fault Tolerant (PBFT) [19], Quorum/Update (QU) [22], Hybrid Quorum (HQ) [48], Zyzzyva [18], FaB [49], Spinning [50], Robust BFT SMR [51], and Aliph [52]. Because of its wide used in Blockchain systems, here, we explain PBFT protocol [19]. we will also detail Zyzzyva protocol [18] as we have applied it in designing our protocol.

- PBFT – Practical Byzantine Fault Tolerance (PBFT) protocol, proposed by Castro *et al.* [19], is one of the efficient algorithm to reach consensus in the presence of Byzantine participants in a partially synchronous network. It requires $3f + 1$ nodes in the system to be able to tolerate up to f byzantine nodes. Since it is designed for partially synchronous network, the nodes are coordinated in rounds. Each round comprises of three phases, namely, *pre-prepare*, *prepare*, and *commit* [19].
  - Pre-prepare – in this phase the primary node (current leader), triggers the next round (in which the nodes must agree upon a value m) by sending a *"pre-prepare"* message to other nodes. When nodes receive the *"pre-prepare"* message, they check the validity of the m. If m is valid and correct, then, they proceed to the next phase, *prepare*.
  - Prepare – every node sends a *"prepare"* message to other nodes. A node that has received a quorum of $2f + 1$ *"prepare"* message for the value m, it proceeds to the next phase, *commit*.
  - Commit – when a node is in this phase, it sends a *"commit"* message to other nodes. Once, nodes received a quorum of $2f + 1$ *"commit"* message for the value m, they are assured that they are in the safe state as enough members have acknowledged and recorded the decision for m. Thus, they update their state by adding m.

  These phases are the normal operations of the PBFT, when the primary is correct and reliable. However, if the primary is suspected faulty, then the protocol falls into a sub-protocol, *View-Change*. When a node notices a malicious behaviour of primary (of the current view), or stuck in one phase for a long time with no progress, it initiates a view-change and stops performing the current view's operations. If $2f + 1$ nodes triggered the view-change phase, then the next primary takes over. PBFT has been applied in some existing blockchain systems such as Hyperledger fabric [53], Tendermint [54]. However, it has some drawbacks. (1), all nodes in the system are fixed, and predefined (before the protocol starts). Thus, it fails to provide the join/leave property (which is one of the essential properties for public blockchains), (2), it has a $O(n^2)$ communication complexity, due to the nodes communicating directly to all other nodes [3]. This causes significant delay in the networks with large number of nodes.
- Zyzzyva – In 2007 Kotla *et al.* [18] proposed Zyzzyva, a fast BFT consensus protocol based on state machine replication. Zyzzyva applied a speculative approach in updating the state which resulted in a high throughput [21], [23]. In Zyzzyva, when the primary (current leader) receives the client's request it sends it to other nodes. Nodes respond to the request without first running the expensive three-phase commit protocol to reach agreement. It comprises of two paths [18], [21], [23]. One is a *two-phase* path which resembles the PBFT protocol. The other is the *fast* path which has removed the *commit* phase from the PBFT protocol. In the fast path, the state is updated once the client receives $3f + 1$ *prepare* message. However, if there is not enough $(3f + 1)$ *commit* messages, then the protocol falls into the two-phase path to guarantee the progress. We explain them in more details below.
  - *Fast path* – this path does not have commit message. When a client receives the $3f + 1$ *prepare* messages from the nodes, it commits the message. This is an optimistic approach which may fail. To guarantee the progress, Zyzzyva proposes a second path, *two-phase* path which resembles the PBFT.
  - *Two-phase path* – If the client receives between $2f + 1$ and $3f$ *prepare* messages, then the client needs to collect $2f + 1$ *commit* messages. Thus, the client creates a *commit-certificate* and sends it to the nodes. The nodes send the *commit* message to the client. If client receives $2f + 1$ *commit* messages, then the request is complete and client commits the message.

  Zyzzyva has a view-change sub-protocol which will be initiated if a primary is faulty.

Performance analysis provided in [18], [21], shows that Zyzzyva has significantly improved performance compare to the previous BFT protocols, namely, PBFT [19] and QU [22]. Zyzzyva'a latency has reached the lower bound. And the

throughput overhead of Zyzzyva has reduced remarkably. We believe this fundamental improvement can notably benefit blockchain protocols if applied correctly. Thus, we applied Zyzzyva, to propose a new blockchain consensus protocol to reduce the latency and increase the throughput of the current protocols.

APPENDIX B–SCHNORR SIGNATURE

It is a digital signature scheme which is defined over an Elliptic curve and generates the key pairs similar to ECDSA algorithm, i.e. x is the secret key, which is a random integer from $\{1..n-1\}$, where n is the order of the subgroup, and public key is $pk \leftarrow x \times G$, where G is the base point of the elliptic curve. To generate the signature, a random number R, which is a point over the curve, is generated by $R \leftarrow k \times G$, where k is a random integer. Signature s is $s = k + h(pk, R, m).x$, where h is a standard hash function. Signature is valid if the $s \times G = R + h(pk, R, m) \times pk$. Schonrr signature can be used to build multi party signatures, m-of-n multi signature, however, it needs to use some kind of merkle tree to be able to satisfy the multi signature requirements, i.e. including the public keys of all the participants in signing. The drawback of this scheme is when the size of m and n become large, the merkle tree size blows up exponentially. This signature is applied in CoSi protocol by applying spanning trees to manage the public keys. We have applied CoSi protocol to design our protocol.

ACKNOWLEDGMENT

The authors would like to thank the ARC for their support to this research through the Australian Government Research Training Program Scholarship.

REFERENCES

[1] N. Sohrabi and Z. Tari, "On the scalability of blockchain systems," in *Proc. IEEE Int. Conf. Cloud Eng. (IC2E)*, Apr. 2020, pp. 1–12.
[2] I. Eyal, A. E. Gencer, E. G. Sirer, and R. Van Renesse, "Bitcoin-NG: A scalable blockchain protocol," in *Proc. USENIX Symp. Networked Syst. Design Implement. (NSDI)*, 2016, pp. 45–59.
[3] E. K. Kogias, P. Jovanovic, N. Gailly, I. Khoffi, L. Gasser, and B. Ford, "Enhancing bitcoin security and performance with strong consistency via collective signing," in *Proc. USENIX*, 2016, pp. 279–296.
[4] L. Luu, V. Narayanan, C. Zheng, K. Baweja, S. Gilbert, and P. Saxena, "A secure sharding protocol for open blockchains," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2016, pp. 17–30.
[5] E. Kokoris-Kogias, P. Jovanovic, L. Gasser, N. Gailly, E. Syta, and B. Ford, "OmniLedger: A secure, scale-out, decentralized ledger via sharding," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2018, pp. 583–598.
[6] M. Zamani, M. Movahedi, and M. Raykova, "RapidChain: Scaling blockchain via full sharding," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Jan. 2018, pp. 931–948. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3243734.3243853
[7] T. Crain, C. Natoli, and V. Gramoli, "Evaluating the red belly blockchain," 2018, *arXiv:1812.11747*. [Online]. Available: http://arxiv.org/abs/1812.11747
[8] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich, "Algorand," in *Proc. 26th Symp. Operating Syst. Princ.*, Oct. 2017, pp. 51–68.
[9] S. Popov, "IOTA Whitepaper v1.4.3," *New Yorker*, vol. 81, no. 8, pp. 1–28, 2018. [Online]. Available: https://assets.ctfassets.net/r1dr6vzfxhev/2t4uxvsIqk0EUau6g2sw0g/45eae33637ca92f85dd9f4a3a-218e1ec/iota1_4_3.pdf
[10] A. Churyumov. (2018). *Byteball*. pp. 1–49. [Online]. Available: https://byteball.org/
[11] C. Lemahieu, "Nano: A feeless distributed cryptocurrency network," White Paper 8, 2018.
[12] D. Schwartz, N. Youngs, and A. Britto, "The ripple protocol consensus algorithm," Ripple Labs Inc., White Paper 5, no. 8, 2014.
[13] B. Chase and E. MacBrough, "Analysis of the XRP ledger consensus protocol," 2018, *arXiv:1802.07242*. [Online]. Available: http://arxiv.org/abs/1802.07242
[14] A. M. Antonopoulos, *Mastering Bitcoin*. Newton, MA, USA: O'Reilly, 2017.
[15] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," *Self-Published Paper*, 2008. [Online]. Available: https://bitcoin.org/bitcoin.pdf
[16] Y. Xiao, N. Zhang, W. Lou, and Y. T. Hou, "A survey of distributed consensus protocols for blockchain networks," *IEEE Commun. Surveys Tuts.*, vol. 22, no. 2, pp. 1432–1465, 2nd Quart., 2020.
[17] S. Bano, A. Sonnino, M. Al-Bassam, S. Azouvi, P. McCorry, S. Meiklejohn, and G. Danezis, "Sok: Consensus in the age of blockchains," 2017, *arXiv:1711.03936*. [Online]. Available: http://arxiv.org/abs/1711.03936
[18] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, "Zyzzyva: Speculative Byzantine fault tolerance," in *Proc. SOSP*, vol. 27, no. 4, 2007, pp. 45–58.
[19] M. Castro and B. Liskov, "Practical Byzantine fault tolerance," in *Proc. OSDI*, vol. 99, 1999, pp. 173–186.
[20] E. Syta, I. Tamas, D. Visher, D. I. Wolinsky, P. Jovanovic, L. Gasser, N. Gailly, I. Khoffi, and B. Ford, "Keeping authorities,' honest or bust' with decentralized witness cosigning," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2016, pp. 526–545.
[21] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, "Zyzzyva: Speculative Byzantine fault tolerance," *ACM Trans. Comput. Syst.*, vol. 27, no. 4, pp. 1–39, Dec. 2009.
[22] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. K. Reiter, and J. J. Wylie, "Fault-scalable Byzantine fault-tolerant services," in *Proc. 20th ACM Symp. Oper. Syst. Princ. (SOSP)*, 2005, pp. 59–74.
[23] I. Abraham, G. Gueta, D. Malkhi, L. Alvisi, R. Kotla, and J.-P. Martin, "Revisiting fast practical byzantine fault tolerance," 2017, *arXiv:1712.01367*. [Online]. Available: http://arxiv.org/abs/1712.01367
[24] J. R. Douceur, "The sybil attack," in *Proc. Int. Workshop Peer-to-Peer Syst.* (Lecture Notes in Computer Science), vol. 2429, 2002, pp. 251–260.
[25] E. Syta, P. Jovanovic, E. K. Kogias, N. Gailly, L. Gasser, I. Khoffi, M. J. Fischer, and B. Ford, "Scalable bias-resistant distributed randomness," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2017, pp. 444–460.
[26] S. Micali, M. Rabin, and S. Vadhan, "Verifiable random functions," in *Proc. 40th Annu. Symp. Found. Comput. Sci.*, 1999, pp. 120–130.
[27] B. Awerbuch and C. Scheideler, "Towards a scalable and robust DHT," *Theory Comput. Syst.*, vol. 45, no. 2, pp. 234–260, Aug. 2009.
[28] S. Sen and M. J. Freedman, "Commensal cuckoo: Secure group partitioning for large-scale services," *ACM SIGOPS Oper. Syst. Rev.*, vol. 46, no. 1, pp. 33–39, Feb. 2012. [Online]. Available: http://dl.acm.org/citation.cfm?id=2146389
[29] I. Abraham, S. Devadas, D. Dolev, K. Nayak, and L. Ren, "Efficient synchronous Byzantine consensus," 2017, *arXiv:1704.02397*. [Online]. Available: http://arxiv.org/abs/1704.02397
[30] N. Alon, H. Kaplan, M. Krivelevich, D. Malkhi, and J. Stern, "Scalable secure storage when half the system is faulty," in *Proc. Int. Colloq. Automata, Lang., Program.* (Lecture Notes in Computer Science), vol. 1853, 2000, pp. 576–587.
[31] N. Alon, H. Kaplan, M. Krivelevich, D. Malkhi, and J. Stern, "Addendum to 'Scalable secure storage when half the system is faulty' [Inform. Comput. 174 (2)(2002) 203-213]," *Inf. Comput.*, vol. 205, no. 7, pp. 1114–1116, 2007.
[32] P. Maymounkov and D. Mazieres, "Kademlia: A peer-to-peer information system based on the XOR metric," in *Proc. Int. Workshop Peer-to-Peer Syst.* Cham, Switzerland: Springer, 2002, pp. 53–65.
[33] H. Dang, T. T. A. Dinh, D. Loghin, E.-C. Chang, Q. Lin, and B. C. Ooi, "Towards scaling blockchain systems via sharding," in *Proc. Int. Conf. Manage. Data (SIGMOD)*, 2019, pp. 123–140.
[34] Y. Xiao, N. Zhang, J. Li, W. Lou, and Y. T. Hou, "Distributed consensus protocols and algorithms," in *Blockchain for Distributed Systems Security*. Hoboken, NJ, USA: Wiley, 2019, pp. 25–50.
[35] G. Coulouris, J. Dollimore, T. Kindberg, and G. Blair, *Distributed Systems: Concepts and Design*. London, U.K.: Pearson, no. 5, 2015. [Online]. Available: http://repositorio.unan.edu.ni/2986/1/5624.pdf
[36] A. S. Tanenbaum and M. Van Steen, *Distributed Systems: Principles and Paradigms*. Upper Saddle River, NJ, USA: Prentice-Hall, 2007.

[37] H. Attiya, A. Bar-Noy, and D. Dolev, "Sharing memory robustly in message-passing systems," *J. ACM*, vol. 42, no. 1, pp. 124–142, Jan. 1995.
[38] M. Kamble, M. T. Shinde, M. Kothiwale N, and K. S. S, "Real time and distributed computing systems," *IOSR J. Comput. Eng.*, pp. 53–56.
[39] M. J. Fischer, "The consensus problem in unreliable distributed systems (a brief survey)," in *Proc. Int. Conf. Fundam. Comput. Theory*, 1983, pp. 127–140.
[40] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem," *ACM Trans. Program. Lang. Syst.*, vol. 4, no. 3, pp. 382–401, Jul. 1982.
[41] B. Alpern and F. B. Schneider, "Defining liveness," *Inf. Process. Lett.*, vol. 21, no. 4, pp. 181–185, Oct. 1985.
[42] H. Attiya and J. Welch, *Distributed Computing: Fundamentals, Simulations, and Advanced Topics*, vol. 19. Hoboken, NJ, USA: Wiley, 2004.
[43] 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.
[44] C. Cachin and M. Vukolić, "Blockchain consensus protocols in the wild," in *Proc. Leibniz Int. Informat.*, vol. 91, 2017, pp. 1–24.
[45] C. Natoli, J. Yu, V. Gramoli, and P. Esteves-Verissimo, "Deconstructing blockchains: A comprehensive survey on consensus, membership and structure," 2019, *arXiv:1908.08316*. [Online]. Available: http://arxiv.org/abs/1908.08316
[46] L. Lamport, "The part-time parliament," *ACM Trans. Comput. Syst.*, vol. 16, no. 2, pp. 133–169, May 1998.
[47] B. M. Oki, and B. H. Liskov, "Viewstamped replication: A new primary copy method to support highly-available distributed systems," in *Proc. 7th Annu. ACM Symp. Princ. Distrib. Comput.*, 1988, pp. 8–17.
[48] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira, "HQ replication: A hybrid quorum protocol for Byzantine fault tolerance," in *Proc. 7th USENIX Symp. Oper. Syst. Design Implement. (OSDI)*, 2006, pp. 177–190.
[49] J.-P. Martin and L. Alvisi, "Fast byzantine consensus," *IEEE Trans. Depend. Sec. Comput.*, vol. 3, no. 3, pp. 202–215, Jul. 2006.
[50] G. S. Veronese, M. Correia, A. N. Bessani, and L. C. Lung, "Spin One's wheels? Byzantine fault tolerance with a spinning primary," in *Proc. 28th IEEE Int. Symp. Reliable Distrib. Syst.*, Sep. 2009, pp. 135–144.
[51] A. Clement, E. Wong, L. Alvisi, M. Dahlin, and M. Marchetti, "Aardvark: Making Byzantine fault tolerant systems tolerate Byzantine faults," in *Proc. Symp. Quart. J. Mod. Foreign Literatures*, 2009, pp. 153–168. [Online]. Available: http://portal.acm.org/citation.cfm?id=1558988
[52] P. L. Aublin, R. Guerraoui, N. Knežević, V. Quéma, and M. Vukolić, "The next 700 BFT protocols," *ACM Trans. Comput. Syst.*, vol. 32, no. 4, pp. 363–376, 2015.
[53] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, and S. Muralidharan, "Hyperledger fabric: A distributed operating system for permissioned blockchains," in *Proc. 13th EuroSys Conf.*, Apr. 2018, pp. 1–15.
[54] J. Kwon, "TenderMint: Consensus without mining," *Blockchain.Com*, vol. 6, pp. 1–10, 2014. [Online]. Available: https://tendermint.com/static/docs/tendermint.pdf
[55] M. Pease, R. Shostak, and L. Lamport, "Reaching agreement in the presence of faults," *J. ACM*, vol. 27, no. 2, pp. 228–234, Apr. 1980.

NASRIN SOHRABI received the degree (Hons.) in computer science from RMIT University, Australia, in 2018, and the bachelor's degree in computer software engineering from Islamic Azad University, Iran. She is currently pursuing the Ph.D. degree in computer science with RMIT University. She also works on the security (access controls) and privacy of the data on edge and cloud environments. Her research interest includes scalability and performance of the blockchain systems.

ZAHIR TARI received the bachelor's degree in mathematics from the University of Algiers (USTHB), Algeria, in 1984, and the M.Sc. degree in operational research and the Ph.D. degree in computer science from the University of Grenoble, France, in 1985 and 1989, respectively. He is currently a Full Professor in distributed systems with RMIT University, Australia. He is the coauthor of six books (John Wiley, Springer) and he has edited over 25 conference proceedings. His research interests include system's performance (e.g., P2P, cloud, and IoT) as well as system's security (e.g., SCADA, SmartGrid, cloud, and IoT). He was a recipient of over 11M$ in funding from the Australian Research Council (ARC) and lately part of a successful 7th Framework Australia to European (AU2EU) bid on Authorisation and Authentication for Entrusted Unions. He is an Associate Editor of *ACM Computing Surveys*, the IEEE TRANSACTIONS ON COMPUTERS (TC), the IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS (TPDS), and the *IEEE Cloud Computing Magazine*.