SAZyzz: Scaling AZyzzyva to Meet Blockchain Requirements

Nasrin Sohrabi[a,*], Gauthier Voron[b], Vincent Gramoli[b], Zahir Tari[a], Jakapan Suaboot[a]

[a]*Centre of Cyber Security Research & Innovation (CCSRI), School of Computing Technologies, RMIT University, Melbourne, Australia*
[b]*School of Computer Science, Sydney University, Sydney, Australia*

Abstract

We present SAZyzz, a leader-based Byzantine Fault Tolerant consensus protocol for partially synchronous networks. SAZyzz exhibits a better performance/scalability compared to the state-of-the-art leader-based BFT consensus protocols. It is built on top of AZyzzyva (i.e., the protocol that addressed the Zyzzyva's safety violation). The tree-based communication model adopted in SAZyzz enables the protocol to enhance the scalability of AZyzzyva. Additionally, it reduces the communication complexity to $O(\log N)$ in two paths of the protocol.

The tree-based topology has been however argued to incur limitations when used to designing BFT consensus protocols. This is due to a strong assumption tree-based protocols are built upon, where the internal nodes of a tree are required not to be Byzantine, which leads to a trade-off between tolerating Byzantine faults and better performance/scalability. This paper shows that, with the current technological infrastructures available for industrial systems, such as Trusted Execution Environment (TEE) and Public Key Infrastructure (PKI), this assumption is realistic.

SAZyzz comprises of fast-path and backup-path, each of which has two modes: *simple mode* and *scalable mode*. We first attempted to remove the strong honest client assumption of AZyzzyva (as it can compromise the entire system if a client is malicious) by involving the primary in making commit decisions. This modification was introduced as the *simple mode* in SAZyzz. Then, we applied the tree-based communication model to address the scalability issue by introducing the *scalable mode* to the design. To reach a consensus in a scalable way, SZAzyzz has therefore the followings: Fast Path Simple Mode *(FPSiM)*, Fast Path Scalable Mode *(FPSCaM)*, Slow Path Simple Mode *(SPSiM)*, and Slow Path Scalable Mode *(SPSCaM)*. To show the efficiency and feasibility of SAZyzz's adoption for blockchain systems, we also designed and implemented the ZyConChain blockchain system based on SAZyzz. The evaluation results provided in this paper show that SAZyzz can significantly improve the performance/scalability of blockchain systems.

Keywords: Blockchain, Distributed Computing, Consensus Algorithms, Security

1. Introduction

The emergence of the blockchain technology paved a new way on designing decentralized and distributed systems, thereby creating a new line of research for both academia and industry. Based on the extensive studies conducted on this technology, *scalability* has been identified as a major limitation of blockchain systems [10, 23, 21, 25, 8, 44, 28, 42, 9, 35, 38, 37, 20] that researchers have attempted to address. This fundamental issue has been rooted in the consensus component of the blockchain, which is a core component of the technology. Despite the significant work carried out in this area, blockchain's scalability is still far from meeting the expectations of today's data intensive generic applications. Hence, this research aims to address the bottleneck by proposing a scalable consensus protocol. Specifically, we focus on one of the existing consensus algorithms, called AZyzzyva [7], and improve its scalability to become suitable for blockchain protocols.

The consensus problem is well-known and well-studied in distributed systems. The problem is that replicas (i.e., the nodes in a given system) require to agree on a data value (e.g., a block of data) while some of these replicas might be faulty or unreliable by experiencing either a *crash failure* or a *Byzantine failure* [40, 24, 41, 32, 17]. Hence, to reach an agreement among these potentially faulty nodes, the system requires a consensus protocol that is Crash Fault Tolerant (*CFT*) or Byzantine Fault Tolerant (*BFT*). Despite few exceptions (e.g., [20]), one of the major drawbacks in consensus protocols is that, when the number of replicas increases, the system's performance degrades tremendously.

Over the last four decades, several solutions have been proposed to solve the consensus problem. These solutions are categorized into three types based on the network synchrony model [22, 9, 41, 40, 6], i.e., synchronous network, partially synchronous network, and asynchronous network. In this work, we focus on partially synchronous networks, where there is an unknown bounded time Δ for message delivery. This means messages are guaranteed to be delivered but there is an unknown delay for delivery. In this model, if F replicas are Byzantine, then $N \geq 3F + 1$ replicas are required to agree on the same requests. The algorithms proposed for the consensus problem in partially synchronous network include DLS [22], Paxos [31],

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

view stamped replication (VR) [36], Practical Byzantine Fault Tolerant (PBFT) [15], Quorum/Update (QU) [1], Hybrid Quorum (HQ) [18], Zyzzyva [29], FaB [34], Spinning [39], Aliph [7], DBFT [19], ByzArchipelago [5] and Hotstuff [43]. Among these protocols, some follow a leader-based model (e.g., PBFT, Zyzzyva, and Hotsfuff) and some follow a leaderless model (e.g., DBFT, ByzArchipelago). In this study we further narrow down the scope of the research to leader-based BFT consensus algorithms in partially synchronous networks, as these are more common. It is worth mentioning that Crain et al. [20] have recently successfully applied DBFT to scale the Red Belly blockchain, a comparison with this protocol will however be addressed in our future work.

Most of the existing BFT leader-based consensus protocols have *scalability* issues. We define *scalability* as the ability of a given system to increase the number of replicas and still being able to reach a consensus with a reasonable/acceptable performance. Current protocols are restricted in terms of the number of replicas needed to reach an agreement with an acceptable performance (i.e., the use of an upper bound threshold x for replicas in a network to reach an agreement). If the network size (i.e., # replicas in a system) exceeds x, then the overall performance declines significantly; and in some cases, the protocol might even break and fail to come to an agreement. This research aims:

- To increase the threshold x (the maximum number of replicas where the protocol can reach agreement).
- To increase the maximum throughput that the system can offer at x replicas.

We start by examining existing consensus algorithms, identify their drawbacks, and then attempt to obviate the identified limitations. Studies [3, 29] showed that among the BFT (leader-based) consensus algorithms for partially synchronous networks, Zyzzyva [29] offers the highest throughput compared to other protocols (e.g., PBFT [29, 3]), and this is why we started our study with the Zyzzyva algorithm. As however reported in [2], further studies showed that Zyzzyva has a safety issue (i.e., safety violation). In 2010, Aublin et al. introduced an abstraction model (for BFT consensus algorithms) to simplify the implementation of consensus algorithms. In their paper, they proposed/developed AZyzzyva which mimics the behaviour of Zyzzyva in the best-case situation (for which Zyzzyva was optimized). AZyzzyva, unlike Zyzzyva does not have the safety violation issue [2]. We thus focused our studies on AZyzzyva, and it became our starting point for designing the proposed protocol. Our studies on AZyzzyva showed that this protocol has major limitations preventing it from being used in real systems. Indeed, similar to other consensus algorithms, AZyzzyva does not scale. Additionally, AZyzzyva relies on a strong assumption, which is similar to Zyzzyva, i.e., honest client. The protocol can break if the client is malicious.

We address here AZyzzyva's bottleneck issue so to enable the protocol to scale to large networks while offering a better performance compared to the state-of-the-art Hotstuff protocol. This newly designed scalable AZyzzyva, called **SAZyzz**, provides the following contributions:

- To enable SAZyzz to increase the threshold x (i.e., to increase the number of replicas in the system), we applied a tree-based communication model for the message communication between the primary replica (the leader) and the other replicas. This model has been added to the protocol by introducing two additional paths to AZyzzyva's original paths (AZyzzyva is explained in detail in §6). Thus, SAZyzz comprises the following four paths: ***F**ast **P**ath **Si**mple **M**ode* **(*FPSiM*)** (explained in §3.1), ***F**ast **P**ath **Sca**lable **M**ode* **(*FPSCaM*)** (explained in §3.2), ***S**low **P**ath **Si**mple **M**ode* **(*SPSiM*)** (explained in §3.3), and ***S**low **P**ath **Sca**lable **M**ode* **(*SPSCaM*)** (explained in §3.4).
- We relaxed the strong honest client assumption by adding the client's task to the primary replica as well. This has been achieved by adding one extra task to the replicas, i.e., sending their responses to both primary and client. This enables the system to detect when the client is not honest.
- The fixed primary model from AZyzzyva is also removed, i.e., SAZyzz relies on changing the primary on an epoch based manner (that is the primary is changing in each epoch).
- We fully designed and implemented SAZyzz in Java (jdk-14.0.1) with roughly 25K lines of code. Evaluation results are shown in §5.
- We applied SAZyzz in a blockchain context, i.e. in ZyConChain [38] and designed and implemented this blockchain system. Refer to §4 and §5.3 for details.

2. The Model

We consider a system consisting of a fixed set of $N = 3F + 1$ replicas, where every replica is indexed by $i \in [N]$ and $[N] = \{1, ..., N\}$. Among these replicas, a set of $f \subset [N]$, from 1 up to $|F|$, are Byzantine. The replicas are connected in a peer-to-peer model. We also assume here a partially synchronous network, i.e., messages will eventually be received by the replicas with an unknown delay τ. The proposed model does not have restriction on the client to be honest or malicious, i.e., the client can be either honest or malicious. However, we assume that the client cannot collude with the primary. Table 1 summarises the notation used in this paper.

2.1. Tree Structure

We applied a tree-based communication model for structuring the network (in two paths of the protocol, explained later) to enhance performance/scalability. There have been debates on applying tree-based communication to design BFT consensus algorithms. The tree-based model imposes a limitation to

Notation	Definition
τ	Network delay
sk	Private key
pk	Public key
σ	Signature
$a\sigma$	Aggregated signatures
apk	Aggregated public key
m	Message
$H()$	Hash function
H(Block)	Hash of the block
N	Total number of replicas
F	Number of faulty replicas
FPSiM	Fast Path Simple Mode
FPSCaM	Fast Path Scalable Mode
SPSiM	Slow Path Simple Mode
SPSCaM	Slow Path Scalable Mode

Table 1: Notations

the protocol as it requires to consider an assumption: the internal nodes of the tree must not be Byzantine for the information to propagate top-down. In other words, all the byzantine nodes must be located in the leaf nodes. This can be viewed as a trade-off between tolerating Byzantine faults and performance/scalability.

The assumption about Byzantine failures considered in BFT protocols is however unnecessarily strong for the current technological settings. Indeed, with the current available technological infrastructures (e.g., Trusted Execution Environment (TEE), public key infrastructure (PKI) with the certificate mechanisms), the design of consensus protocols does not necessarily need to consider/tolerate the f Byzantine nodes. Usually the faults in a current infrastructure are more likely to be a crash fault. The recent works, namely FastBFT [33] and CheapBFT [26], leveraged the current advances in the TEE technology and redesigned the consensus algorithms to reduce communication costs.

Thus, the following two assumptions are introduced for our proposal:

- We assume all the internal nodes of the tree to be correct nodes, i.e., not Byzantine. Hence, the messages, which include consensus messages and blocks, are propagated over the tree to the leaf nodes.
- During the gossip protocol, we assume that all the internal nodes are correct (i.e., not Byzantine) and transactions are propagated to the leaf nodes.

As depicted in Figure 1, the tree is a binary tree in which the root is the primary and other nodes of the tree are the replicas. When a new leader is elected, the tree is constructed at each replica by putting the primary replica as the root of the tree and all the replicas create a similar tree. To propagate a message, the primary sends the message down to its children, and then each child node sends the same message down to its children and so forth, until the message reaches the leaf nodes. Upon receiving the message, the leaf nodes sign it and send their votes for the message back to their parents in a bottom-up fashion. Once the primary collects all the responses for the message, it creates the final aggregated signature for the requested message.

Instead of sending a message to all the nodes, which has a communication cost of $O(N)$, the primary sends it to only two nodes (its immediate children). Hence, the tree reduces the communication costs to $O(\log N)$. This helps increase the number of nodes in the network without adding bottlenecks on the primary, because the primary sends the message only to two nodes regardless of the total number of replicas in the network. This also has a drawback as it increases the latency, but we believe this drawback is negligible compared to the scalability improvements the technique provides.

2.2. Cryptographic primitives

SAZyzz uses a multi-party signature scheme [12, 13], namely BLS multi-signature [14, 13], which aggregates all the digital signatures in a common message. It also aggregates the public keys. For the signature verification, this scheme only needs to verify a short multi-signature, which is faster than verifying several signatures separately. Thus, each SAZyzz's replica is associated a pair of (sk_i, pk_i) key that is generated using the following BLS function $KeyGen()$:

- $\alpha_i \xleftarrow{R} \mathbb{Z}_q$ (choose at random).
- $h_i \leftarrow g_1^{\alpha_i} \in \mathbb{G}_1$.
- output $pk_i = (h_i)$ and $sk_i = (\alpha_i)$.

To sign a message m received from the primary replica (i.e., the leader), replicas use the following BLS $Sign(sk_i, m)$ function:

- output $\sigma_i \leftarrow H_0(m)_q^{\alpha_i} \in \mathbb{G}_0$.

The replicas then will send their signatures back to the primary replica. Upon receiving the signatures from the replicas, the primary replica aggregates all the signatures using the following aggregation function:

- $Aggregate((pk_1, \alpha_1), ..., (pk_n, \alpha_n))$:
  - compute $(t_1, t_2, ..., t_n) \leftarrow H_1(pk_1, ..., pk_n) \in R^n$.
  - output the multi-signature $\alpha \leftarrow \alpha_1^{t_1}, ..., \alpha_n^{t_n} \in \mathbb{G}_0$.

Then the primary replica sends the triple (α, m, apk), where α is the aggregated signature, m is the message, and apk is the aggregated public key computed as follows:

- compute $(t_1, t_2, ..., t_n) \leftarrow H_1(pk_1, ..., pk_n) \in R^n$.
- compute the aggregated public key $apk \leftarrow pk_1^{t_1}, ..., pk_n^{t_n}$.

The replicas can then verify the aggregated signature using the following BLS function:

- $Verify(apk, m, \alpha)$
- if $e(g_1, \alpha) = e(apk, H_0(m))$, output *"accept"*, otherwise output *"reject"*.

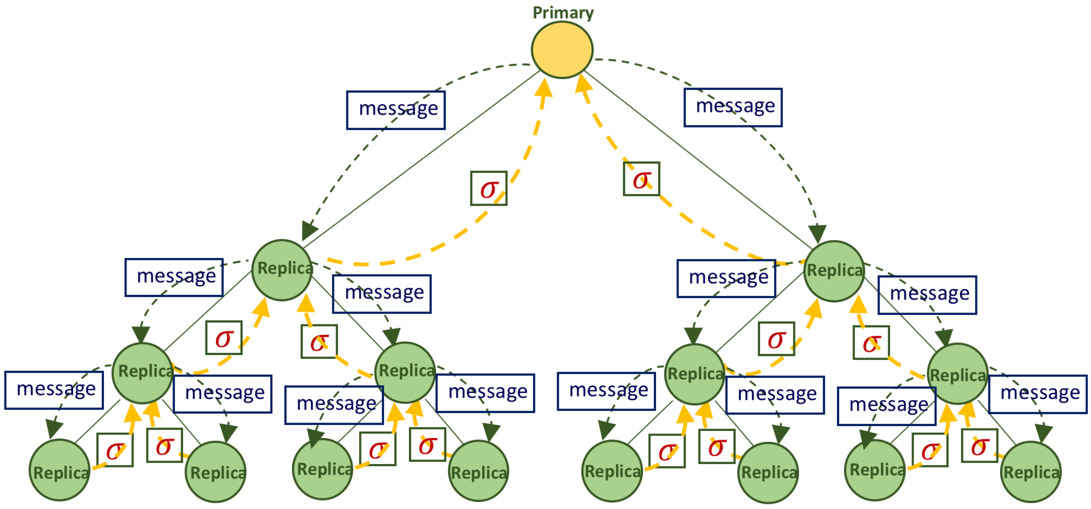

Figure 1: Tree structure

3. Paths

SAZyzz introduces four paths, and each path has different number of phases to reach an agreement. The first two phases of all the paths are similar: SAZyzz (for all the paths) starts by (phase 1) clients sending their requests to one (or more e.g., $(f+1)$) replicas. Then, the second phase (phase 2) is triggered, i.e., replicas who received the clients' requests, broadcast them (using gossip protocol [27]) to their neighbours such that the entire network receive the requests. Note that the difference between phase 1 of SAZyzz and AZyzzyva is that in AZyzzyva, the client sends the request to the primary, however, in SAZyzz, client can send its request to any replica. Then we added the phase 2 to disseminate the request to other replicas including the primary. We have added this phase to enable the protocol to work in the epoch based model when it is needed, e.g., in the blockchain context.

3.1. FPSiM Path

Details

Referring to Figure 2, in phase 3, the primary creates a proposal (i.e., a block) and broadcasts (through reliable broadcast [16]) it to all the replicas in the network. Replicas, upon receiving the block, which includes the client request, need to execute the client's request and verify (and vote for) the block, phase 4. If the proposal is verified then replicas *"accept"* the block by signing the proposal message, which contains $H(block)$ (i.e., the hash of the block) and message m, i.e., the string *"BLOCK-PROPOSAL"*. For the signing, replicas use the $Sign()$ function explained in §2.2. The replicas, then, (*I*) send their responses (i.e., the response from the execution of the client's request) directly to the client, and (*II*) send their signatures to the primary. If the client receives $(3f+1)$ similar responses from replicas, then the request can be committed. Meanwhile, the primary collects all the votes from the replicas. If $(3f+1)$ signatures (including the primary's signature) are collected, then the proposal (block) is accepted and committed. The primary then aggregates the collected signatures, using the $Aggregate()$ function explained in §2.2. Then, it creates the *COMMIT* message, which includes 1) the $H(block)$, i.e., the hash of newly accepted block, 2) message m, i.e., the string "*COMMIT*", and 3) $a\sigma$, i.e., the aggregated signatures, and sends it to all the replicas, this is phase 5. At this stage, the agreement is reached and one round of the consensus is terminated. The extra task, i.e., the replicas sending their votes back to the primary, is added to detect the malicious client. Let us now explain how it detects it: consider that the replicas send their responses back to the client and their votes back to the primary. The malicious client, despite receiving $(3f+1)$ same responses from the replicas, sends a "*PANIC*" message to the replicas. If the primary sends the "*COMMIT*" message to the replicas, then the honest replicas can detect the mismatch between the two messages received from the primary and the client. Thus, the honest replicas detect the malicious behaviour and inform the network.

Communication Complexity

The communication complexity of this path is computed by the summation of the communication cost of each phase, which is $O(1)$ in phase 1 and $O(n)$ in all the remaining phases, namely, phases 2, 3, 4, and 5. Thus, the total communication complexity is computed as follows, $O(1)+O(n)+O(n)+O(n)+O(n) = O(1)+4\times O(n) \simeq O(n)$.

Contribution

FPSiM removes the strong assumption of AZyzzyva's honest client by modifying the decision maker for *COMMIT* phase of fast path of AZyzzyva. In AZyzzyva, the client, upon receiving $(3f+1)$ same results from replicas, decides on commit, which requires the strong assumption of client being honest. However, this decision in FPSiM is finalized by both the client and the primary, thus it relaxes the assumption.

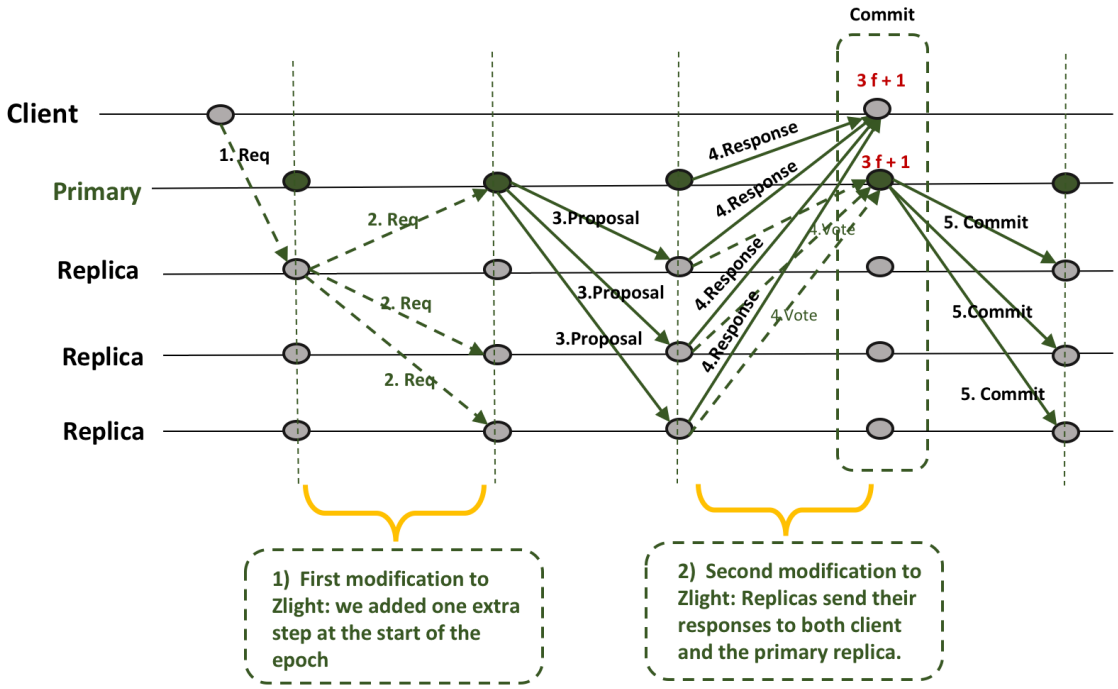

Figure 2: FPSiM (Fast Path Simple Mode) Phases

Result

FPSiM scales better and provides significantly higher throughput in comparison to the recently proposed leader-based consensus protocol, namely Hotstuff [43], with the same settings and assumptions. The experimental results provided in §5 demonstrate this improvement. Furthermore, the protocol does not rely on honest clients, which makes the protocol secure against malicious client attacks (i.e., when the client sends different decisions to different replicas).

3.2. FPSCaM Path

Details

Improving scalability has been our main concern/goal in devising SAZyzz. Thus, after relaxing the honest client assumption, we then switched our attention to further enhancing the scalability and performance of the protocol. We applied a tree based communication model for the communications between the primary and other replicas. This reduces the amount of messages that the primary needs to send (through the limited bandwidth) to the replicas, as instead of passing the messages to all the replicas, the primary only sends them to its children. Then each child node sends the messages to their immediate children until the messages reach the leaf nodes. Then the leaf nodes vote for the messages and send their signatures back up on the tree to their parents.

As depicted in Figure 3, after phase 2 the primary creates a new proposal (i.e., block) and sends it to the network based on the tree communication model (i.e., sends the proposal to its children, and then the children send it down to the tree until it reaches the leaf nodes): this is phase 3. Phase 4 is then triggered, that is the leaf nodes (I) execute the client's request, and (II) sign the proposal using the $Sign()$ function of §2.2, if verified. They, then, send their responses to the client and their signatures back up on the tree to their parents. The parent node then aggregates the signatures received from its children with its own signature using the *Aggregate*() function explained in § 2.2, and sends it back up on the tree until it reaches the primary. Upon receiving $3f$ votes from its children, the primary aggregates them with its own signature and then triggers the commit phase (phase 5). During this phase, the primary uses the same tree-based communication to send the *COMMIT* message to the replicas. If a client receives $(3f + 1)$ similar responses from the replicas, and the replicas receive the "*COMMIT*" message from the primary, the request can be committed. At this stage the agreement on the proposal is reached and the one round of the consensus is terminated.

Communication Complexity

The tree-based communication reduces the communication complexity of phases $3, 4$, and 5 of the FPSiM path. Similar to FPSiM, the total complexity (including all the phases) is computed by the sum of the communication costs of all phases, i.e., $O(1)$ in phase 1, $O(n)$ in phase 2 and $O(log(n))$ in all the remaining phases, namely, phase $3, 4$, and 5. Thus, the total communication complexity is as follows: $O(1) + O(n) + O(log(n)) + O(log(n)) + O(log(n)) = O(1) + O(n) + 3 \cdot O(log(n)) \simeq O(n) + O(log(n))$. When removing the cost of phases 1 and 2 from both paths, one can see the improvements of the cost in the phases $3, 4$, and 5. This cost is $O(n)$ for FPSiM; however, this is reduced to $O(log(n))$ in FPSCaM path.

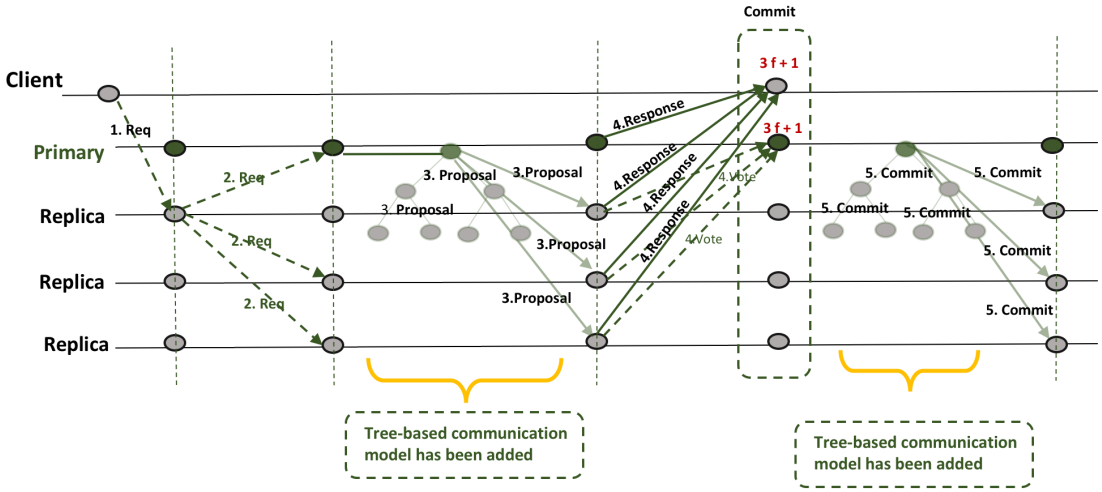

Figure 3: FPSCaM (Fast Path-Scalable Mode) Phases

Contribution

The main contribution of this path is the tree-based communication that is added to the FPSiM. Thus, compared to the AZyzzyva consensus protocol, this path has a main contribution, i.e., the tree-based communication.

Result

This enables the system to increase the number of the nodes in the network without degrading the performance of the system when comparing to existing solutions.

3.3. SPSiM Path

Details

Referring again to FPSiM and FPSCaM, if one (or more) of the replicas is faulty, the protocol cannot reach an agreement, i.e. the consensus does not terminate. To address this problem, similar to AZyzzyva, SAZyzz relies on a backup path, i.e., the slow path mode. This path is triggered if the client or the primary does not receive enough similar responses and enough signatures from the replicas within a period T (i.e., when the client sends a request it sets a timer that expires after a period T), respectively. Let us assume that in phase 4 of the FPSiM path the primary either receives less than $3f$ signatures (within T, i.e., $(2f + 1) <$ # Signatures $< 3f$), or the client receives less than $(3f + 1)$ similar responses from the replicas within the period T. Thus, the client and the primary send a "*PANIC*" message to the replicas (phase 5), and the primary also aggregates the received signatures and sends it along with the "*PANIC*" message. Upon receiving this message, the replicas stop executing requests and send their history along with their signatures to the primary and client. The replicas will then send "*ABORT*" message for the subsequent messages. This is when the system falls back to the backup path.

AZyzzyva applies the PBFT [15] protocol as it is backup solution. Following this, we also apply PBFT. However, any other BFT consensus algorithm can be replaced for the backup model. Once the consensus is reached based on the PBFT protocol, the response is sent back to the client. Here, the agreement is reached and one round of the consensus is terminated.

Communication Complexity

For computing the communication cost of this path we first compute the cost from phase 1 to phase 6, i.e., before entering the backup solution. We will not add the communication cost of the backup algorithm here, only because this can vary depending on the choice of the backup algorithm. Thus, the complexity cost is computed in the same way as FPSiM, namely by the summation of the communication costs of each phase, which is here $O(1)$ for phase 1 and $O(n)$ for phases 2, 3, 4, 5, and 6. Thus, the total communication complexity is $O(1)+O(n)+O(n)+O(n)+O(n)+O(n)+O(n) = O(1)+5\cdot\times O(n) \simeq O(n)$. If we assume the cost of the backup algorithm is X, with X not necessarily linear, then the final cost is $\simeq O(n) + X$.

Contribution

Compared to AZyzzyva, the main contribution of this path is the removal of honest client assumption from deciding on switching to the fall-back path of AZyzzyva. When the AZyzzyva's client does not receive enough, $(3f + 1)$, same responses from the replicas, it sends the *PANIC* message to all replicas, including the primary. Here, the replicas need to receive the same *PANIC* message from the primary to accept the switching.

Result

Since the primary replica has become part of the decision making task, the system can detect the fault and misbehaviour easily compared to when the client is responsible to decide.

3.4. SPSCaM Path

Details

This path is the scalable version of the SPSiM path. SPScaM uses the tree-based communication model (applied in FPSCaM)

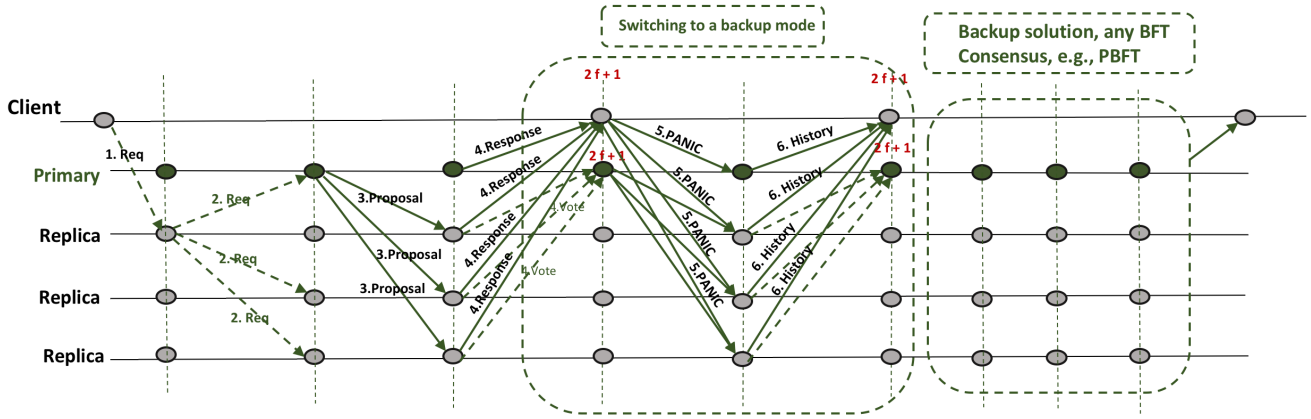

Figure 4: SPSiM (Slow Path-Simple Mode) Phases

to make the fall back path (SPSiM) scalable. Thus, if the primary does not receive the required signatures (i.e., $(3f + 1)$) in the fast paths, then it can fall into this path instead of the SPSiM path.

The path is depicted in Figure 5. This path is similar to the SPSiM path with one difference, i.e., all the communication between the primary and the replicas are over the tree. Additionally, we applied the tree communication in our backup algorithm, i.e., PBFT. Hence, in phase 3, the primary sends the proposal to the replicas based on the tree communication, explained before. Consequently, the replicas reply back to the primary in a bottom-up fashion over the tree. Similarly, in phase 5 the primary sends the *PANIC* message over the tree and the replicas reply back to the primary by sending their responses to their parents. Additionally, when the protocol enters the backup path, all the communication between the primary and the replicas are over the tree.

Communication Complexity

Similar to SPSiM, we compute the complexity for the path before entering the backup consensus. Thus, the communication cost is computed in the same way as the FPSCaM path (the summation of each phase). The total communication cost is equal to $O(1) + O(n) + O(log(n)) + O(log(n)) + O(log(n))) = O(1) + O(n) + 3 \cdot O(log(n)) \simeq O(n) + O(log(n))$. If the cost of phases 1 and 2 are discarded, the communication cost would then become $O(log(n))$. If we assume the cost of the backup algorithm is X, where X is not necessarily linear, then the final cost is $\simeq O(log(n)) + X$, which indicates an improvement compared to SPSiM, where the cost is $O(n) + X$.

Contribution

The main contribution of this path is the tree-communication model added to SPSiM.

Result

The tree-based communication enables the system to scale to a large number of nodes without compromising its performance. Furthermore, the protocol does not rely on the honest client, which makes the protocol secure against the malicious client attack (i.e., when the client sends different decisions to different replicas to break the system).

To summarise this section, we have introduced four paths in SAZyzz. Two paths are fast-path mode (i.e., FPSiM and FPSCaM) and the other two paths are the fall back mode (i.e., SPSiM and SPSCaM). FPSCam and SPSCaM are the scalable versions of FPSiM and SPSiM, respectively. When applying SAZyzz, only one path of each mode needs to be used. That is, depending on the application requirements (e.g., if scalability is needed), then the scalable version of each mode will need to be applied (i.e., FPSCaM for the fast mode and SPSCaM for the fall back mode).

4. Blockchain Adopting SAZyzz

After devising SAZyzz and evaluating its performance (see § 5), we have used it to design a new blockchain system called ZyConChain [38] (see Figure 6). ZyConChain 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 targeted/modified both the blockchain structure component as well as the consensus component. For the consensus protocol, we used the SAZyzz algorithm. For the structure component, we modified the one block one chain structure. We introduced three types of blocks, namely *parentBlock*, *sideBlock*, and *stateBlock*. These blocks form three separate chains: *main chain*, *sideBlock chain* and *state block chain*, respectively.

Briefly, ZyConChain uses the sharding technique; and thus it divides the network into small groups, *committee*, and introduces the three types of chains, mentioned above. Within each committee nodes process transactions and generate sideBlocks based on the SAZyzz consensus algorithm. The primary of the committee collects transactions from the transaction pool and includes them into a sideBlock. It then triggers the SAZyzz consensus algorithm to reach an agreement amongst the com-

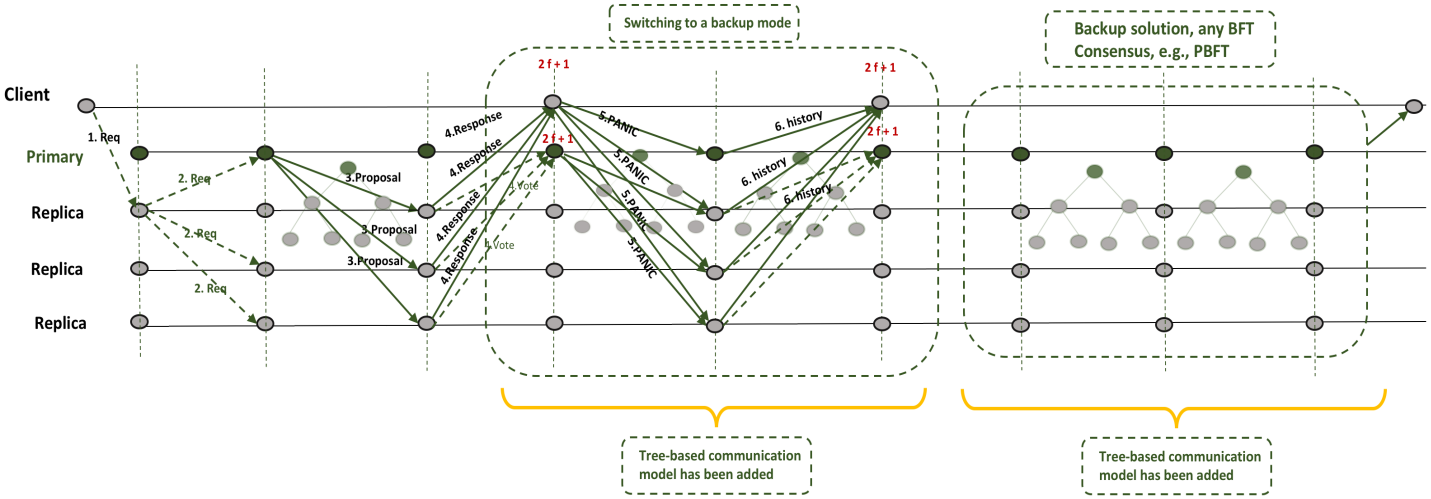

Figure 5: SPScaM (Slow Path-Scalable Mode) Phases

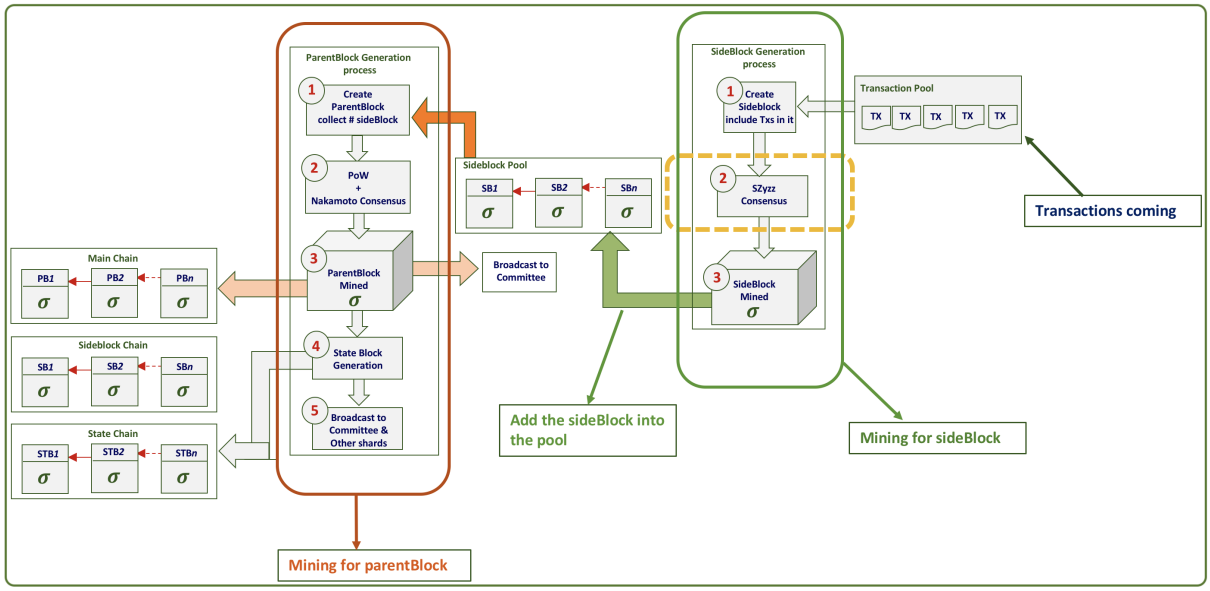

Figure 6: ZyConChain Overview

mittee group for the proposed sideBlock. If the committee agrees on the sideBlock, this will be attached into the sideBlock pool.

ParentBlocks and stateBlocks are also generated within each committee, separately. The leader (primary) generates the parentBlock and stateBlock. To generate the parentBlock, the leader first collects several confirmed sideBlocks from the sideBlock pool and includes them into a parentBlock. Then, it computes the hash for the parentBlock and requests the committee group to reach an agreement for the proposed parentBlock. Once the committee agrees on the parentBlock, this is included into the main chain. Then, the sideBlocks are added and attached to the parentBlock into the sideBlock chain. Finally, the leader creates the stateBlock, which contains the information about the latest update of the main chain and sideBlock chain.

As ZyConChain used the sharding technique, it was required to address the cross-shard transactions limitations associated with this protocol. To address this issue, ZyConChain generates verifiable objects that contain information about cross-shard transactions. The verifiable objects are included into the stateBlock and sent to other committees. Once other shards receive the stateBlock, they will be able to verify the cross-shard transactions confirmation and finalize them. Please refer to [38] for more details about the original work on ZyConChain.

5. Evaluation

We fully implemented SAZyzz as a library in Java (jdk-14.0.1) with roughly 25K lines of code. It contains 11 packages and 146 classes. We carefully followed the well-known software engineering principles during the full implementation of SAZyzz. Specifically, we carefully designed the architecture of the packages to ensure the final product (i.e., the SAZyzz consensus library) has both maintainability and reusability features. Hence, most of the SAZyzz's packages are reusable, and accessible through the library's github.

This section first provides the performance (throughput) and scalability evaluation of SAZyzz and compares it with the state-of-the-art Hotstuff [43]. The experimental results conducted on the SAZyzz based blockchain (i.e., ZyConChain) are then presented.

5.1. Setup

The experiments were conducted on the Victorian Cloud infrastructure (i.e., Nectar) using m3.medium instances. Each instance had 4 vCPUs, 8GB RAM, and 30GB storage, with all running Ubuntu 20.4 & 19.10 OS. All instances were located in the same zone, melbourne-qh2. We ran each replica on a single VM instance, similarly to Hotstuff. We did not modify the bandwidth of the network. The network latency between two machines was $\simeq$ 1 ms.

The implementation of SAZyzz uses $secp256k1$ for all digital signatures. For the hash function, we used $sha256$ for hashing all the messages. There exist two implementations for BLS, namely, *Blst* and *mikilu*. We added both in our implementation. However, Blst is used for the experiments. All results for SAZyzz reflect end-to-end measurement from primary.

For the Hotstuff, we used the provided library on github [1] and setup the clients, replicas and servers on different VM machines for the experiments. The results reflect the end-to-end measurement from clients.

Metrics

We measured the throughput and scalability with two metrics, *Block Size* and *Network Size*, i.e., the number of the total replicas in the network. For the former, we conducted the experiments for sizes in the set {38, 70, 89, 100, 165, 200, 300, 400} (in KB). And for the latter, we varied the size to have the network with size from the set {4, 7, 10, 13, 16, 19, 25, 34, 40, 46}.

Dataset

To test the protocol in a real world setting, we used Ethereum transactions to perform one set of the experiments. We extracted the Ethereum dataset from BigQuery API from google cloud platform *("https://console.cloud.google.com/bigquery?project=tidal-memento-290909&page=jobs")* which is available publicly. We used the following SQL query to download the dataset.

SELECT $from\_address$, $to\_address$, $value$,
$block\_number$, $block\_hash$
FROM "$bigquery - public - data.ethereum\_blockchain.transactions$" AS transactions
WHERE DATE ($block\_timestamp$) BETWEEN
"2018 – 09 – 28" AND "2020 – 09 – 28" LIMIT 1000

5.2. Performance

SAZyzz Results

We first measured throughput in a setting commonly seen in the evaluation of other BFT replication systems. We ran 4 replicas in a configuration that tolerates a single failure, i.e., $f = 1$, while varied the block size. This benchmark used empty (zero-size) transactions (client requests) with no view-change triggered. We then varied the network size with the configuration accordance with that size, e.g., for the network size 7 replicas the configuration tolerates 2 failures. Similarly, we increased the network size up to 46 replicas, where the system tolerates 15 failures.

Figure 7 depicts the results of 4 block sizes, 100, 200, 300, and 400, for 5 different network size, 4, 7, 10, 25, and 46 replicas. When the block size is set to 100, the highest throughput of SAZyzz is around 35K transactions per second, where the network size is 4. When the block size is enlarged, the throughput of the system increases progressively. With the block size 400 KB, the system performs at its highest capacity, where it processes approximately 140K transactions per second for the network size 4. This number is around 40K transactions per second for the network with 46 replicas.

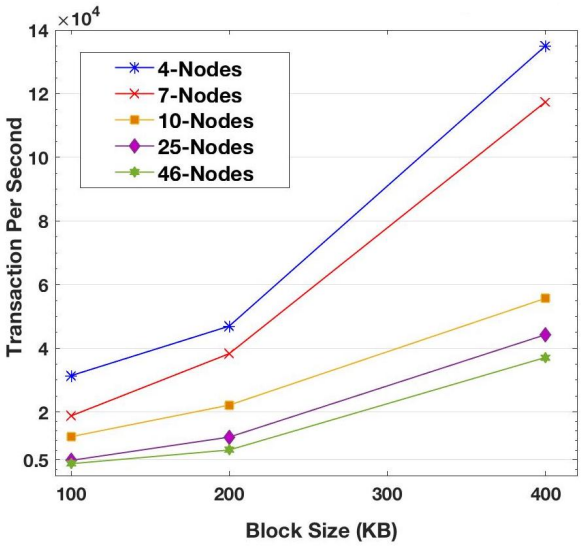

Figure 7: SAZyzz Performance Results

Hotstuff Results

We measured the throughput of Hotstuff for the same configuration. That is, we varied the network size (4, 7, 10, 25, and 46

[1] https://github.com/hot-stuff/libhotstuff

replicas) and block size (100 and 400 KB) and ran the experiments for the empty (zero-size) transactions. Figure 8 shows the results for Hotstuff: it also illustrates the comparison between the two systems, SAZyzz and Hotstuff. When the network size is 4 Hotstuff performs better than SAZyzz, where for block size 100 KB Hotstuff processes around 140K transactions per second while SAZyzz processes 35K transactions. For the block size 400 KB, Hotstuff throughput is roughly 160K, but for SAZyzz, this number is approximately 140K transactions per second. When the number of replicas in the network increases, we can see that SAZyzz performs better than Hotstuff. For instance, when the network comprises of 7 replicas, Hotstuff can process around 65K transactions per second (for block size 400 KB), whereas SAZyzz processes close to 120K for the same block size. Similarly, for the network size 25, Hotstuff throughput is nearly 20K transactions per second but SAZyzz's throughput is over 45K transactions. When the number of replicas is increased to more than 25, Hotstuff failed to process any transaction. This was surprising, as it differs from the results reported by the original work on Hotstuff [43]. We believe this might be due to the infrastructure used for the experiments. Specifically, Hotstuff conducted the experiments on the Amazon EC2 $c5.4xlarge$ instances which had more computational resources available (i.e., each instance had 16 vCPUs supported by Intel Xeon Platinum 8000 processors). However, our computational resources were somehow limited, and details provided in Section § 5.1.

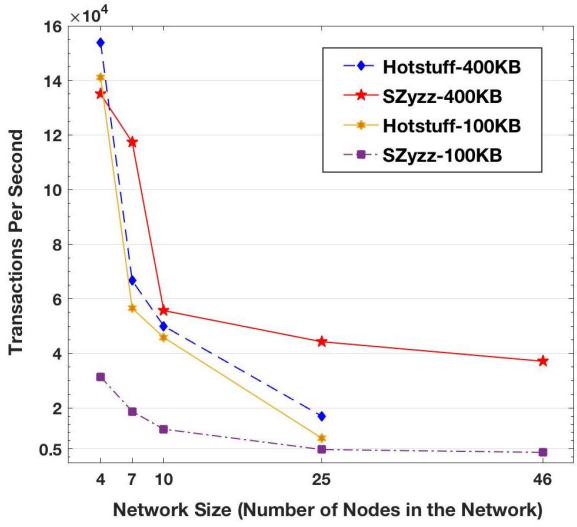

Figure 8: HotStuff & SAZyzz Comparison Results

5.3. Performance Results of ZyConChain

We fully designed and implemented ZyConChain [38] based on SAZyzz consensus library. To evaluate its performance, we conducted several experiments in different configurations. For the baseline, we used Ethereum transactions and set the block size to 38 KB while varying the network size. Second, we increased the block size to 70, 89, and 165 KB and carried out the experiments with the same transactions while changing the network size. For each experiment, we ran the test 5 times to show the standard deviations as error bars. Note that, the results include transaction verification and the BLS multiparty signature verification that are computationally expensive. We would also like to mention that Ethereum transactions are not executed in ZyConChain. We only send them to network to have transactions with a different payload size.

As shown in Figure 10, when the block size increases, ZyConChain performance improves. For block size 38 KB, ZyConChain can process around 1500 transactions per second. When the block size is increased to 165 KB, the throughput is increased to 4k transactions per second. When the network size is increased, we can still see a performance improvement when the block size increases, as depicted in Figure 10. These results indicate that the ZyConChain's performance scales better compared to the state-of-the-art blockchain systems, as depicted in Figure 11. ZyConChain outperformed Rapidchain (when block size is 32 MB), ByzCoin, Omniledger (when its block size is 1 MB), Elastico, Ethereum, and Bitcoin. It is worth mentioning that unlike ZyConChain, these blockchain systems assume a synchronized network while we do not. When comparing however with Omniledger (in the 16 MB block size), ZyConChain processes less transactions per second. Note that the numbers used in Figure 11 are the reported numbers. It is indeed truly challenging to re-implement all the benchmarked protocols, and we therefore only used the data reported in the corresponding papers. We used however the same experimental setting applied in these protocols to provide a possible/plausible comparison between some of these blockchain systems.

5.4. Scalability

To evaluate the scalability of SAZyzz in various dimensions, we carried out three sets of experiments. For the baseline, we used empty transactions with block size set to 100 KB while varying the number of replicas. Second, we increased block size to 200, 300, and 400 for the empty transactions while changing the number of replicas. Third, we applied SAZyzz in a blockchain context, i.e., ZyConChain, and used real world transactions (see §5.1) for the experiments. In this experiment, we varied the block size and the number of replicas. We repeated each run 5 times for each setting to indicate the standard deviations and to show the error bars.

The first experiments, which are depicted in Figure 7, indicate that the throughput (the number of transaction per second) of SAZyzz improves when the block size increases even when the network size is scaled out, i.e., the number of replicas increases. Additionally, as shown in Figure 8, we observed that SAZyzz's performance scales better compared to Hotstuff. For the block size 400 KB and the network size 10 replicas, Hotstuff processed 50K, whereas SAZyzz processed close to 60K transactions per second. When the number of replicas was increased to 25, the performance of Hotstuff deteriorated noticeably compared to SAZyzz. Hotstuff processed less than 20K transactions per second, while SAZyzz processed around 50K transactions per second.

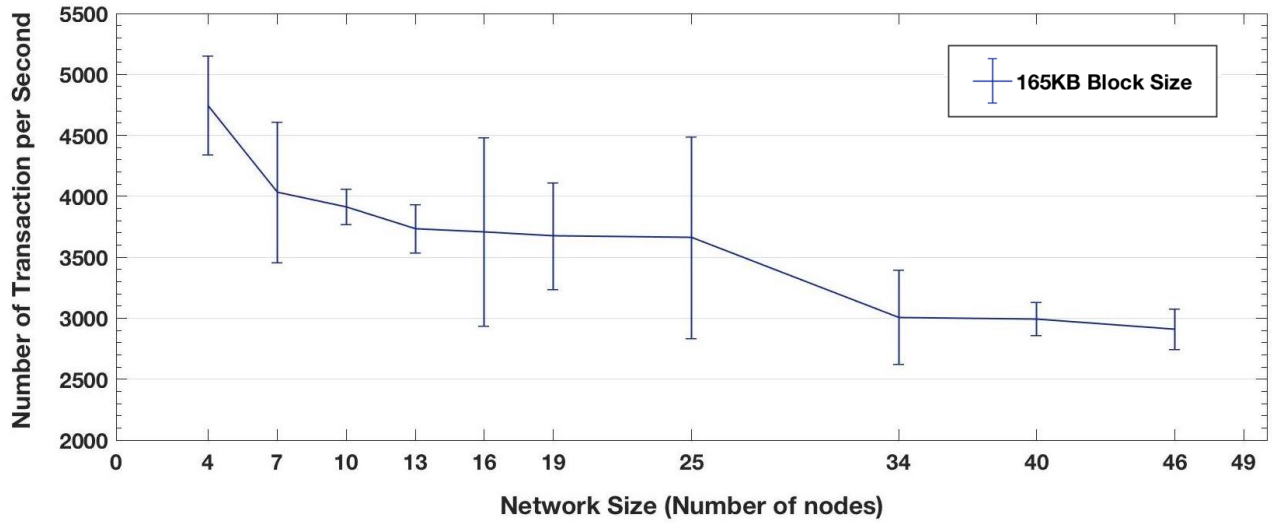

Figure 9: ZyConChain performance in different network size

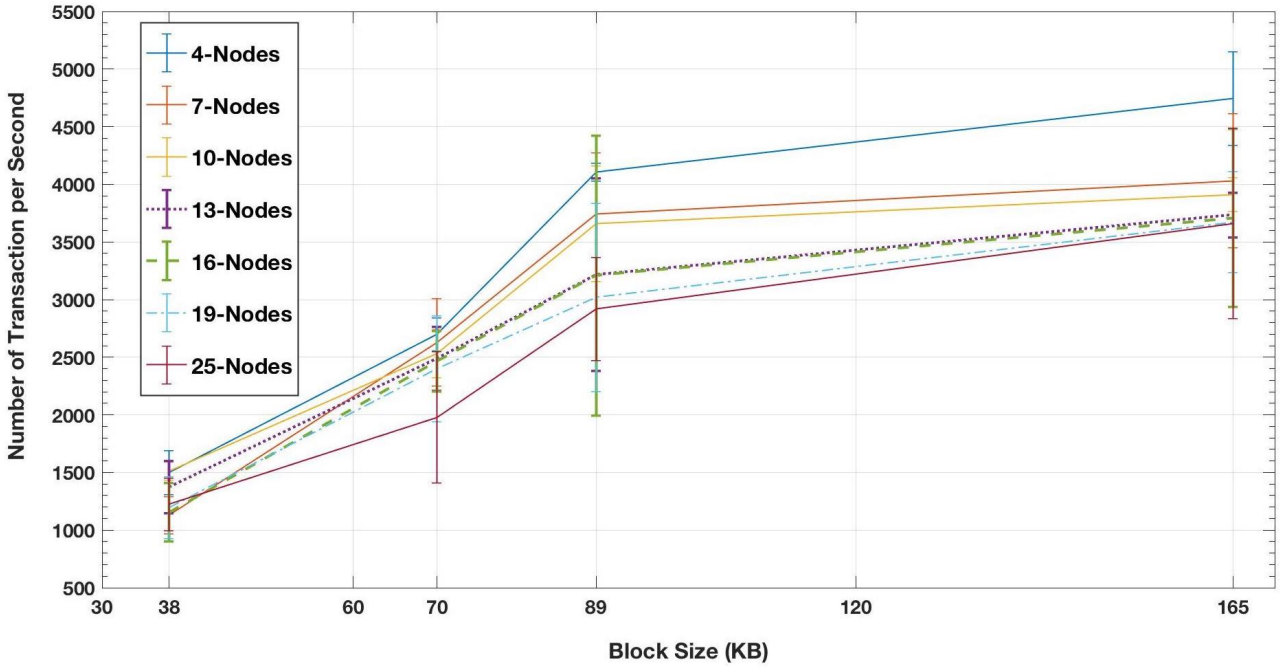

Figure 10: Performance of ZyConChain with different block size

6. Related Work

There has been over four decades of contributions to the solving the consensus problem. This section studies only four solutions, specifically four BFT consensus protocols, each being a representative of one decade. We first discuss the DLS [22] protocol proposed in 1988, and then describe the PBFT [15] protocol proposed in 1999. Later, we provide details of the Zyzzyva [29] protocol, which was proposed in 2007. Since Zyzzyva presents a safety violation, we finally studied the AZyzzyva [7] protocol. Finally, we describe the most recent consensus algorithm, namely Hotstuff [43], proposed in 2019. We will show how these protocols operate as well as their limitations.

DLS

In 1988, Dwork, Lynch, and Stockmeyer[22], initiated the theoretical foundation of partially synchronous consensus algorithms. They divided the consensus problem into *safety* and *liveness* (also known as termination) problems. They, then, formally demonstrated that the consensus problem was solvable in four models, i.e., Crash Fault Tolerance (CFT), omission-

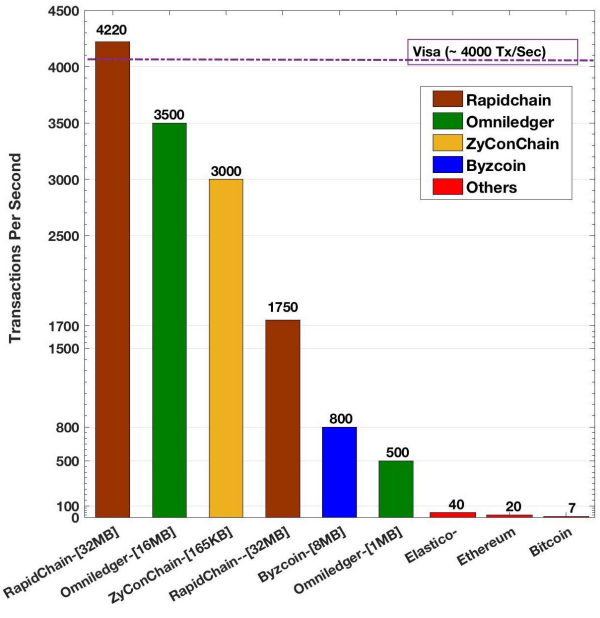

Figure 11: Benchmarking Blockchain Systems

tolerance, Byzantine Fault Tolerance (BFT), unauthenticated BFT, within the Δ synchronous setting.

To achieve agreement in this setting, DLS relies on a broadcast primitive in a round based model. That is an arbitrary replica p (i.e., the proposer) starts the broadcast primitive, which consists of two initial rounds followed by subsequent iterative rounds. The rounds are organized into alternating *"trying"* and *"lock-release"* phases. For the BFT setting the DLS rounds work as follows:

- Each round begins with each of the replicas sending the value v (i.e., the value that they believe is correct) to the p (the round's proposer).
- p *"proposes"* a value v if at least $N-F$ replicas have sent that value to p. (if there is more than one possible value that p can propose, then it will choose one arbitrarily.)
- Once a replica receives the proposed value from p, it locks on the value v and sends an acknowledgment to the p.
- If p receives messages from $2F+1$ replicas that they locked on the value v, p commits that as the final value.

DLS requires the network of $N = 3F + 1$ replicas, where F is the number of Byzantine faults in the network.

Message complexity (i.e., the message transmission in each single round) in DLS is $O(N^4)$. We have achieved a better complexity.

PBFT

The Practical Byzantine Fault Tolerance (PBFT) protocol, proposed by Castro et al. [15], is one of the algorithms that reaches consensus in the presence of Byzantine participants in a partially synchronous network. It requires $(3F + 1)$ replicas 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 three phases, namely, *pre-prepare*, *prepare*, and *commit* [15].

- Pre-prepare: during this phase, the primary replica (the current leader) triggers the next round (in which the replicas must agree upon a value m) by sending a *"pre-prepare"* message to other replicas. When replicas receive the *"pre-prepare"* message, they check the validity of m. If m is valid and correct, then they proceed to the next phase, i.e. *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, proceeds to the next phase, called the *commit* phase.
- 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 a 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 protocol when the primary is correct and reliable. However, if the primary is suspected to be faulty, then the protocol reverts to a sub-protocol, called a *View-Change*. When a node notices a non-responsive or malicious behaviour of the primary (of the current view), 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.

Although PBFT has correctly addressed the consensus problem in the presence of Byzantine nodes, it has two major drawbacks. (1) PBFT's performance was not designed to scale to wide area networks: when the number of replicas in the network increases, the performance of the system degrades. (2) By design, PBFT follows leader-based pattern with an all-to-all communication model, needing a communication complexity of $O(n^4)$ to commit a constant number of transactions as explained elsewhere [20]. This causes significant delays in the network when a large number of nodes is used.

PBFT has led to several other leader-based protocols, namely BFT-SMaRT and Zyzzyva [11, 29]. It has been adopted in some of the existing blockchain protocols as well, such as Hyperledger Fabric [4], and [30] is a variant of it. All these protocols inherit some of the limitations of the PBFT protocol.

Zyzzyva

In 2007, Zyzzyva Kotla et al. [29] was proposed as a fast BFT consensus protocol based on state machine replication. They applied a speculative approach in updating the states, resulting in a high throughput protocol [2, 3]. In this protocol, when the primary (the current leader) receives the clients' requests, it sends them to other replicas. Replicas directly respond to

the requests without first reaching an agreement that requires running the expensive three-phase commit protocol.

Zyzzyva comprises two paths [29, 2, 3]. One is a *two-phase* path that resembles the PBFT protocol. The other one is the *fast* path which bypasses the *commit* phase of the PBFT protocol. In the fast path, the state is updated once the client receives $(3F + 1)$ *prepare* messages. However, if there are not enough $(3F + 1)$ *commit* messages, then the protocol falls back into the two-phase path to guarantee the progress. We explain them in more details below.

- *Fast path*: this path does not have a commit message phase. When a client receives $(3F + 1)$ *prepare* messages from the replicas, it commits the message. This is an optimistic approach that can fail. To guarantee the progress, Zyzzyva proposes a second path, called the *two-phase* path, which resembles to the PBFT's one.
- *Two-phase path*: If the client receives between $(2F + 1)$ and $3F$ *prepare* messages, then it needs to collect $(2F+1)$ *commit* messages. Thus, the client creates a *commit-certificate* message and sends it to the replicas. The replicas reply to the client by sending the *commit* message. If the client receives $(2F + 1)$ *commit* messages, then the request is complete and the client commits the message.

The performance analysis in [3, 29] shows that Zyzzyva has significantly improved performance compared to the previous BFT protocols, such as PBFT [15] and QU [1]. Additionally, Zyzzyva's latency was shown the lowest and its throughput overhead is remarkably low.

Although Zyzzyva has provided substantial performance improvements, it has some major drawbacks that hinder the usage/deployment of this protocol. Indeed, Zyzzyva used the strong honest client assumption. Another downside of Zyzzyva, similar to other BFT consensus algorithms, is its inability to scale when the number of replicas increases. Finally, Zyzzyva has a view-change sub-protocol which will is initiated if the primary is slow or faulty. This protocol is very complex and difficult to implement.

AZyzzyva

In 2010, the authors of [7] proposed AZyzzyva. The main goal of AZyzzyva was to decouple the handling of the "fast-path" and the "slow path" of Zyzzyva, as it was shown that the transition between the two paths in Zyzzyva is extremely complex to implement. The authors introduced two Abstract implementations namely, *ZLight* and *Backup* to address the issue. ZLight is the Abstract that guarantees the progress of the fast-path of Zyzzyva. The Backup is the Abstract that guarantees the safety of the protocol.

ZLight Abstract commits requests when (*I*) there are no link failures, and (*II*) no client is Byzantine (this is what we referred to as the honest client assumption, i.e., one of the issues of AZyzzyva). In these settings, ZLight implements Zyzzyva's "fast-path" model. However, when the client does not receive $(3F + 1)$ similar responses from replicas, it sends a "PANIC" message to replicas (unlike Zyzzyva whose client sends the "commit-certificate" message). Upon receiving the "PANIC" message, replicas stop executing requests and send back to the client a signed message that includes their history. When the client receives back $(2F + 1)$ messages which contain the replica's histories, it generates an "abort history" message and switches to the Backup Abstract.

The Backup Abstract is built as a wrapper around any BFT algorithm. It works as follows, it ignores all the requests sent by the underlying BFT protocol until it receives a request that contains a valid init history, i.e., the abort history that is generated by the client in the previous Zlight path, explained above. When it receives the correct init history, it executes all the requests in the init history and sets its current state. Then it starts executing k number of ordered requests received from the BFT protocol. After committing the k^{th} request, Backup aborts all the subsequent requests and returns a signed message containing the sequence of executed requests, as the abort history. Committing only the k ordered requests is to guarantee the liveness of the protocol.

As mentioned before, AZyzzyva, similar to Zyzzyva, relies on the honest client assumption. Thus, if the client is malicious, the protocol is compromised. Additionally, AZyzzyva is limited to a small number of replicas, hence, if the network grows, i.e., the number of replicas increases, the protocol is not able to maintain its performance and even in some cases (when the number of replicas is bigger than a threshold x) the protocol fails to progress.

Hotstuff

Hotstuff Yin et al. [43], proposed in 2019, is one of the most recent leader-based BFT consensus algorithms designed for partially synchronous networks. It outperforms BFT-SMaRT with a linear communication complexity, their leader election mechanism relies on a round-based model.

The protocol reaches consensus in four phases, namely *prepare*, *pre-commit*, *commit* and *decide* phase. It starts by collecting *new-view* messages from replicas for a new leader. When the leader has collected requests from enough replicas, i.e., $(N - F)$, it starts a new view and creates a new proposal based on the received views, and then sends the prepare message (*prepare* phase) to other replicas. After receiving the prepare message from the leader, other replicas first verify the message. If it is verified, then replicas send the prepare vote back to the leader. Upon receiving $(N-F)$ votes from the replicas for the proposal, the leader triggers the *pre-commit* phase. During this phase, the leader combines the votes (received from replicas) into a prepare quorum certificate and then broadcasts it in a pre-commit messages to all replicas. Upon receiving the pre-commit message, the replicas send their votes back to the leader. When the leader receives pre-commit votes, it combines them into a precommitQC and broadcasts it in commit message (*commit* phase) to all replicas. The replicas that receive the commit message lock their current state and send their vote on the commit message back to the leader. Upon receiving the votes for the commit message, it sends the decide message (*decide* phase) to

all replicas. After receiving the decide message, the replicas update the state they have locked in the *commit* phase.

7. Conclusion

This paper described SAZyzz, a scalable Byzantine consensus protocol for partially synchronous networks. SAZyzz is a leader-based protocol, where the leader changes in an epoch based model. It is built on top of AZyzzva and addressed its limitations, namely scalability and the strong honest client assumption. We adopted a tree-based communication model in SAZyzz to enable the protocol to scale when the number of replicas increases. The protocol proposed four paths for reaching consensus, namely $FPSiM$ §3.1, $FPSCaM$ §3.2, $SPSiM$ §3.3, and $SPSCaM$ § 3.4. SAZyzz has removed the honest client assumption from AZyzzyva by giving the client's task to the primary replica as well. The evaluation results showed that SAZyzz's performance scales better compared to the state-of-the-art, Hotstuff protocol.

To conclude the paper, we show the results in Table 2 relating to the complexity analysis of different SAZyzz's paths. The fast path with the adopted tree model, i.e., $FPSCaM$, has improved the communication costs to $O(log(n))$. The fast path without the tree communication, i.e., $FPSiM$, has linear cost. For the backup paths, we can see where we have adopted the tree communication, i.e., $SPSCaM$ path, we have a $O(log(n))$ cost plus the cost of the backup algorithm, shown as X. For the path without the tree, i.e., $SPSiM$, we have a linear communication cost plus the cost of the backup algorithm, thus the communication complexity is $O(log(n)) + X$. When this is compared with Hotstuff, where the communication cost is linear, we can see that SAZyzz with the $FPSCaM$ path outperforms the Hotstuff.

Path	Communication Cost
$FPSiM$	$O(n)$
$FPSCaM$	$O(log(n))$
$SPSiM$	$O(n) + X$
$SPSCaM$	$O(log(n)) + X$

Table 2: Communication Complexity

Acknowledgment

The first and fourth authors of this paper would like to thank the Australian Research Council (ARC) for the support of this work under the DP (Discovery Project) scheme (DP200100005). The scholarship of the first author is provided by this grant.

References

[1] Michael Abd-El-Malek, Gregory R Ganger, Garth R Goodson, Michael K Reiter, and Jay J Wylie. 2005. Fault-scalable Byzantine fault-tolerant services. *ACM SIGOPS Operating Systems Review* 39, 5 (2005), 59–74.

[2] Ittai Abraham, Guy Gueta, Dahlia Malkhi, Lorenzo Alvisi, Rama Kotla, and Jean-Philippe Martin. 2017. Revisiting Fast Practical Byzantine Fault Tolerance. *arXiv:1712.01367 [cs.DC]* i (2017), 1–13. arXiv:1712.01367 `http://arxiv.org/abs/1712.01367`

[3] Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. 2009. Zyzzyva : Speculative Byzantine Fault Tolerance. 27, 4 (2009). `https://doi.org/10.1145/1658357.1658358`

[4] Elli Androulaki, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolić, Artem Barger, Sharon Weed Cocco, Jason Yellick, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, and Gennady Laventman. 2018. Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains Elli. *EuroSys* (2018), 1–15. `https://doi.org/10.1145/3190508.3190538`

[5] Karolos Antoniadis, Antoine Desjardins, Vincent Gramoli, Rachid Guerraoui, and Igor Zablotchi. 2021. Leaderless Consensus. In *Proceedings of the 41st IEEE International Conference on Distributed Computing Systems (ICDCS'21)*.

[6] Hagit Attiya and Jennifer Welch. 2004. *Distributed computing: fundamentals, simulations, and advanced topics*. Vol. 19. John Wiley & Sons.

[7] Pierre Louis Aublin, Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. 2015. The next 700 BFT protocols. *ACM Transactions on Computer Systems* 32, 4 (2015), 363–376. `https://doi.org/10.1145/2658994`

[8] Shehar Bano, Mustafa Al-Bassam, and George Danezis. 2017. The road to scalable blockchain designs. *USENIX; login: magazine* (2017).

[9] Shehar Bano, Alberto Sonnino, Mustafa Al-Bassam, Sarah Azouvi, Patrick McCorry, Sarah Meiklejohn, and George Danezis. 2019. SoK: Consensus in the Age of Blockchains. In *Proceedings of the 1st ACM Conference on Advances in Financial Technologies (AFT '19)*. 183–198.

[10] Simon Barber, Xavier Boyen, Elaine Shi, and Ersin Uzun. 2012. Bitter to better—how to make bitcoin a better currency. In *International conference on financial cryptography and data security*. Springer, 399–414.

[11] Alysson Bessani, Joao Sousa, and Eduardo EP Alchieri. 2014. State machine replication for the masses with BFT-SMART. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 355–362.

[12] Alexandra Boldyreva. 2003. Threshold signatures, multisignatures and blind signatures based on the Gap-Diffie-Hellman-group signature scheme. *PKC* 2567 (2003), 31–46.

[13] Dan Boneh, Manu Drijvers, and Gregory Neven. 2018. Compact multi-signatures for smaller blockchains. In *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 435–464.

[14] Dan Boneh, Ben Lynn, and Hovav Shacham. 2001. Short signatures from the Weil pairing. In *International conference on the theory and application of cryptology and information security*. Springer, 514–532.

[15] Miguel Castro, Barbara Liskov, et al. 1999. Practical Byzantine fault tolerance. 99, 1999 (1999), 173–186.

[16] Jo-Mei Chang and Nicholas F. Maxemchuk. 1984. Reliable broadcast protocols. *ACM Transactions on Computer Systems (TOCS)* 2, 3 (1984), 251–273.

[17] George Coulouris, Jean Dollimore, Tim Kindberg, and Gordon Blair. 2015. *Distributed Systems Concept and Design*. Number 5. pearson.

[18] James Cowling, Daniel Myers, Barbara Liskov, Rodrigo Rodrigues, and Liuba Shrira. 2006. HQ replication: A hybrid quorum protocol for Byzantine fault tolerance. *OSDI 2006 - 7th USENIX Symposium on Operating Systems Design and Implementation* (2006), 177–190.

[19] Tyler Crain, Vincent Gramoli, Mikel Larrea, and Michel Raynal. 2018. Dbft: Efficient leaderless byzantine consensus and its application to blockchains. In *2018 IEEE 17th International Symposium on Network Computing and Applications (NCA)*. IEEE, 1–8.

[20] Tyler Crain, Christopher Natoli, and Vincent Gramoli. 2021. Red Belly: a secure, fair and scalable open blockchain. In *Proceedings of the 42nd IEEE Symposium on Security and Privacy (S&P'21)*.

[21] Kyle Croman, Christian Decker, Ittay Eyal, Adem Efe Gencer, Ari Juels, Ahmed Kosba, Andrew Miller, Prateek Saxena, Elaine Shi, Emin Gün Sirer, et al. 2016. On scaling decentralized blockchains. In *International conference on financial cryptography and data security*. Springer, 106–125.

[22] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. 1988. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)* 35, 2 (1988), 288–323. https://doi.org/10.1145/42282.42283

[23] Ittay; Eyal, Adem; Efe Gencer, Emin; Gun Sirer, and Robbert Van Renesse. 2016. Bitcoin-NG: A Scalable Blockchain Protocol Ittay. *USENIX Symposium on Networked Systems Design and Implementation (NSDI '16)* (2016).

[24] Michael J Fischer. 1983. The consensus problem in unreliable distributed systems (a brief survey). (1983), 127–140.

[25] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. 2017. Algorand. *SOSP'17* (2017), 51–68. https://doi.org/10.1145/3132747.3132757 arXiv:1607.01341

[26] Rüdiger Kapitza, Johannes Behl, Christian Cachin, Tobias Distler, Simon Kuhnle, Seyed Vahid Mohammadi, Wolfgang Schröder-Preikschat, and Klaus Stengel. 2012. CheapBFT: Resource-efficient Byzantine fault tolerance. In *Proceedings of the 7th ACM european conference on Computer Systems*. 295–308.

[27] Richard Karp, Christian Schindelhauer, Scott Shenker, and Berthold Vocking. 2000. Randomized rumor spreading. In *Proceedings 41st Annual Symposium on Foundations of Computer Science*. IEEE, 565–574.

[28] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ewa Syta, and Bryan Ford. 2018. OmniLedger: A Secure, Scale-Out, Decentralized Ledger via Sharding. *Proceedings - IEEE Symposium on Security and Privacy* 2018-May (2018), 583–598. https://doi.org/10.1109/SP.2018.000-5

[29] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. 2007. Zyzzyva: Speculative Byzantine fault tolerance. *SOSP'07* 27, 4 (2007). https://doi.org/10.1145/1658357.1658358

[30] Jae Kwon. 2014. TenderMint : Consensus without Mining. *the-Blockchain.Com* 6 (2014), 1–10. https://tendermint.com/static/docs/tendermint.pdf

[31] Leslie Lamport and Digital Equipment. [n.d.]. The Part-Time Parliament. 16, May 1998 ([n. d.]), 133–169.

[32] Leslie Lamport, Robert Shostak, and Marshall Pease. 1982. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 4, 3 (1982), 382–401. https://doi.org/10.1145/357172.357176

[33] Jian Liu, Wenting Li, Ghassan O Karame, and N Asokan. 2018. Scalable byzantine consensus via hardware-assisted secret sharing. *IEEE Trans. Comput.* 68, 1 (2018), 139–151.

[34] Jean-philippe Martin, Lorenzo Alvisi, and Senior Member. 2006. Fast Byzantine Consensus. 3, 3 (2006), 202–215.

[35] Christopher Natoli, Jiangshan Yu, Vincent Gramoli, and Paulo Esteves-Verissimo. 2019. Deconstructing Blockchains: A Comprehensive Survey on Consensus, Membership and Structure. (2019). arXiv:1908.08316 http://arxiv.org/abs/1908.08316

[36] Brian M. Oki and Barbara H. Liskov. 1988. Viewstamped replication: A new primary copy method to support highly-available distributed systems. *Proceedings of the seventh annual ACM Symposium on Principles of distributed computing* (1988), 8–17.

[37] Nasrin Sohrabi and Zahir Tari. 2020. On the scalability of Blockchain Systems. *IEEE International Conference on Cloud Engineering* (2020), 1–12. https://doi.org/10.1145/1544012.1544020

[38] Nasrin Sohrabi and Zahir Tari. 2020. ZyConChain: A Scalable Blockchain for General Applications. *IEEE Access* 8 (2020), 158893–158910.

[39] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, and Lau Cheuk Lung. 2009. Spin One ' s Wheels ? Byzantine Fault Tolerance with a Spinning Primary. (2009). https://doi.org/10.1109/SRDS.2009.36

[40] Yang Xiao, Ning Zhang, Jin Li, Wenjing Lou, and Y. Thomas Hou. 2019. Distributed Consensus Protocols and Algorithms. *Blockchain for Distributed Systems Security* 25 (2019). https://doi.org/10.1002/9781119519621.ch2

[41] Yang Xiao, Ning Zhang, Wenjing Lou, and Y. Thomas Hou. 2020. A Survey of Distributed Consensus Protocols for Blockchain Networks. *IEEE Communications Surveys & Tutorials* c (2020), 1–1. https://doi.org/10.1109/comst.2020.2969706 arXiv:1904.04098

[42] Cheng Xu, Ce Zhang, and Jianliang Xu. 2019. vchain: Enabling verifiable boolean range queries over blockchain databases. In *Proceedings of the 2019 international conference on management of data*. 141–158.

[43] Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. 2019. Hotstuff: Bft consensus with linearity and responsiveness. *PODC 2019* (2019), 347–356.

[44] Mahdi Zamani, Mahnush Movahedi, and Mariana Raykova. 2018. RapidChain: Scaling Blockchain via Full Sharding Mahdi. *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security - CCS '18* (2018), 931–948. https://doi.org/10.1145/3243734.3243853