CollaChain: A BFT Collaborative Middleware for Decentralized Applications

Submission #185 – Regular paper

***Abstract*—The sharing economy is centralizing services, leading to misuses of the Internet: we can list growing damages of data hacks, global outages, and even uses of data to manipulate their owners. Unfortunately, there is no decentralized web where users can interact peer-to-peer in a secure way. Blockchains incentivize participants to individually validate every transaction and impose their block to the network. As a result, the validation of smart contract requests is computationally intensive while the agreement on a unique state does not make full use of the network.**

In this paper, we propose CollaChain, a new byzantine fault tolerant blockchain compatible with the largest ecosystem of DApps that leverages collaboration. First, the participants executing smart contracts collaborate to validate the transactions, hence halving the number of validations required by modern blockchains (e.g., Ethereum, Libra). Second, the participants in the consensus collaborate to combine their block proposal into a superblock, hence improving throughput as the system grows to hundreds of nodes. In addition, CollaChain offers the possibility to its users to interact securely with each other without downloading the blockchain, hence allowing interactions via mobile devices. CollaChain is effective at outperforming the Concord and Quorum blockchains and its throughput peaks at 4500 TPS under a Twitter DApp (Decentralized Application) workload. Finally, we demonstrate CollaChain's scalability by deploying it on 200 nodes located in 10 countries over 5 continents.

I. Introduction

As the number of misuses of Internet data grows, so does the need for decentralized middleware rewarding individuals for sharing data. These misuses stem from a "centralization" of the web according to Turing-awardee Tim Berners-Lee, who contributed to the design of a decentralized alternative to his 30-year-old Web [67]. At its origin, the Web helped users communicate with each other through their desktop computers. In the 2000s users started sharing data through Google, Facebook, Microsoft, and Amazon. By 2025, the sharing economy—to which users can contribute with their mobile phone—is expected to represent $335 billion [25]. This centralization has severe drawbacks: it exposes data to leaks and hacks [57] and it facilitates user manipulation [74].

Decentralized applications, or *DApps* for short, are increasingly popular at allowing users to trade services peer-to-peer without transferring ownership. In the third quarter of 2020, DApps transaction volume experienced an 8-fold increase reaching $125B [31]. With 96% of this volume occurring on top of the Ethereum blockchain [87] alone, most DApps are, however, plagued by Ethereum capacity capped at ∼15 transactions per second (TPS) [23]. Even without forcing miners to resolve a crypto-puzzle to obtain a proof-of-work, the proof-of-authority alternative of Ethereum delivers ∼80 TPS [63]. Ethereum suffers from (i) an expensive validation during the execution of smart contracts as well as (ii) an incentive for consensus participants to compete into a fierce battle that aims at imposing their block to the rest of the system. As a result, Ethereum already experienced congestion due to DApps [30] and its capacity is inherently too low to support a DApp with the 4000+ TPS throughput of Twitter [64].

In this paper, we propose CollaChain, a new collaborative byzantine fault tolerant middleware designed for decentralizing the sharing economy. It is compatible with the largest ecosystem of DApps as it runs an adapted version of the Ethereum Virtual Machine (EVM), called Scalable EVM (SEVM), and decouples the traditional blockchain design into an (i) SEVM component that divides the validation of EVM nodes by two as the number n of blockchain participants tends to infinity and (ii) a distributed consensus component that potentially decides a number of transactions proportional to n, hence addressing the two aforementioned limitations to scale as n increases. Although a SEVM node also validates transactions eagerly to mitigate denial-of-service (DoS) attacks, we limit the number of nodes receiving this transaction to achieve (i). Consensus participants collaborate to combine their proposal into the same *superblock* decision, as opposed to the competitive classic consensus approach, to achieve (ii).

Perhaps more importantly, a user of CollaChain, who wants to ensure its DApp can access a consistent state of the blockchain without trusting a central entity, does not need to download the blockchain. CollaChain simply requires a web-based or mobile app and leverages the upper-bound $f < n/3$ on the number of arbitrary (byzantine) failures among n servers to reach consensus [62]. Other blockchains typically require users to either download block headers to verify that the current state is consistent, a process called "synchronizing", before issuing requests [40] or trust a central entity, which defeats the blockchain purpose. Ethereum fast synchronization requires more than 280 GB of free storage space and takes 4 hours on average on an i3.2xlarge AWS EC2 instance with 8 vCPUs, 61 GiB memory and 1.9 TiB NVMe SSD [41], a task nearly impossible for any mobile device. One may think of downloading less information with a "light" synchronization, however, the corresponding validation cannot guarantee that the blockchain is correct due to incomplete blockchain records [42].

CollaChain achieves 2K TPS when deployed on 200 machines spread in 10 countries over five continents. As indicated in Table I, it outperforms the non-sharded blockchains that tolerate byzantine failures. Note that we discuss sharding as an orthogonal optimization in §VI. CollaChain's throughput is two orders of magnitude larger than the Ethereum capacity throughput and, although not reported in Table I, one order of magnitude faster than the 172 TPS of the recent SBFT in a world-scale setting [53]. To the best of our knowledge, the non-sharded blockchains that outperform CollaChain are the ones that only tolerate crash failures. We compare CollaChain performance to Quorum and Concord (§V-B), show the impact

of validation reduction on performance using BlockBench [34], illustrate the scalability of CollaChain to hundreds of machines over 5 continents, and demonstrate a 4500 TPS peak throughput when running the Twitter DApp of the DIABLO benchmarking framework [11].

Blockchain	Fault tolerance	Smart contract	TPS
Ethereum v1.x [87]	probabilistic	Solidity	15
Avalanche [76]	probabilistic	C-Chain	1.3K
Algorand [50]	probabilistic	TEAL	1K
Hyperledger Fabric [6]	crash	ChainCode	3K
FastFabric [51]	crash	ChainCode	20K
Cosmos/Ethermint [18]	byzantine	Solidity	438
Burrow [61]	byzantine	Solidity	765
SBFT [53]	byzantine	Solidity	378
Stellar [65]	byzantine	SSC	100
CollaChain	byzantine	Solidity	4.5K

TABLE I
COMPARISON OF (NON-SHARDED) BLOCKCHAINS WITH POTENTIAL DAPP SUPPORT. THE PERFORMANCE OF ETHEREUM WAS TAKEN FROM [23], THE PERFORMANCE OF HYPERLEDGER FABRIC AND FASTFABRIC WAS TAKEN FROM [51] (THEIR BFT ORDERER [79] IS NOT LISTED AS IT DOES NOT MAKE THESE WHOLE BLOCKCHAINS TOLERATE BYZANTINE FAILURES). PEAK SMART CONTRACT PERFORMANCE OF SBFT WAS MEASURED IN A CONTINENT-SCALE SETTING; IT ACTUALLY ACHIEVES 172 TPS IN A WORLD SCALE SETTING [53]. THE THROUGHPUT OF BURROW IS TAKEN FROM [78]. THE PERFORMANCE OF AVALANCHE WAS TAKEN FROM [76], EVEN IF RECENT CLAIMS ANNOUNCED HIGHER VALUES OBTAINED IN UNKNOWN CONDITIONS. "SOLIDITY" INDICATES THAT THE CORRESPONDING BLOCKCHAIN IS COMPATIBLE WITH THE LARGE ECOSYSTEM OF ETHEREUM DAPPS. WE DISCUSS THE PERFORMANCE OF SHARDED BLOCKCHAINS IN SECTION §V-G.

CollaChain builds upon various results. The superblock optimization already appeared in a UTXO-based blockchain [29], however, CollaChain applies it to smart contracts by decoupling their execution and persistent storage into sub-tasks to avoid request losses as we will illustrate in §V-D. Its SEVM nodes receive the requests from the clients and batch them into blocks that are sent to consensus nodes. Similar to an Ethereum [87], [17] server (i.e., miner) or a Libra [8] server (i.e., validator), an SEVM server of CollaChain validates eagerly a transaction upon reception and validates lazily the same transaction upon execution (after the block that contains it is agreed upon). In contrast with these blockchains, an SEVM node does not propagate the transactions to other nodes upon reception from the client, hence reducing the number of eager validations. Similar to byzantine fault tolerant (BFT) replicated state machine protocols [13], [89], the consensus nodes decide on a unique batch of transactions without assuming synchrony as long as less than a third of consensus nodes are byzantine, which is resilient optimal [62].

In the remainder of the paper, we present our motivations and the necessary background (§II), as well as our goals and assumptions (§III). We then present CollaChain (§IV) that we prove correct, and evaluate it in a geodistributed setting and compare it against other blockchains (§V). Finally, we present the related work (§VI) and conclude (§VII). A smart contract to reconfigure the nodes of CollaChain is provided in Appendix A.

II. BACKGROUND AND MOTIVATIONS

a) Decentralized applications rationale: Decentralized applications (DApps) alleviate many problems induced by the centralization of the sharing economy. To mention a few, YouTube exprienced an outage [75] that DTube could have remedied by sharing videos peer-to-peer [36]. Uber drivers feel manipulated by an opaque matching algorithm [70], whereas the DApp counterpart, called Drife, could offer transparency [35]. So what is the performance necessary to implement such a decentralized version of the sharing economy? To answer this question, let us consider Twitter, which is a popular micro-blogging application. Twitter experiences more than 4000 tweets per second on average and its peak demand largely exceeds this number [64]. It is thus crucial for a mainstream decentralized middleware to support thousands of transactions per second. Unfortunately most blockchains cannot (cf. Table I).

b) The redundant validations of Ethereum: Ethereum [87] features the Ethereum Virtual Machine (EVM) that was proposed in part to cope with the limited expressiveness of Bitcoin [71] and to execute DApps written in a Turing complete programming language as *smart contracts*. Go Ethereum, or `geth` for short, is the mostly deployed Ethereum implementation [49]. In order to check that a request (or transaction) is valid, all of the `geth` *servers* (i.e., miners) must validate twice each executed transaction:

- **Eager validation:** This validation occurs upon reception of a new client transaction and checks the nonce value; that the sender account has sufficient balance; that the gas is sufficient to execute the transaction; and the transaction does not exceed the block gas limit, is signed properly and is not oversized. It reduces the effect of denial-of-service (DoS) attacks as an invalid transaction is dropped early. If the transaction is valid, it is propagated to other servers.
- **Lazy validation:** This validation occurs before transactions are executed in a decided block and simply checks the nonce and whether there is enough gas for execution. This lazy validation is necessary to guarantee that transactions in a newly received decided block are indeed valid. The lazy validation is thus less time consuming in `geth` than the eager validation, this is why we focus on reducing the number of eager validations.

This is an overconservative strategy because each to-be-executed transaction of `geth` is validated twice by each server. This is unnecessary as an invalid transaction coming from a byzantine node will either be dropped by lazy validation prior to execution or fail execution and the state reversed if there is an invalidity not checked by the lazy validation.

It is interesting to note, also, that in a system where few replicas are byzantine, there is no need for all servers to validate all transactions twice. We explain in §IV-C how, without reducing security, we reduce the number k of eager validations per server down to k/n to scale to a large system size n.

c) The inefficiency of byzantine fault tolerant consensus: For security reasons, a blockchain must guarantee that nodes agree on a unique block at each index of the chain. To cope with

malicious (or *byzantine*) participants, this requires solving the byzantine fault tolerant (BFT) consensus problem [72], where every non-byzantine or *correct* node eventually decides a value such that no two correct nodes decide differently. Unfortunately, traditional consensus protocols solve this problem by electing a leader node that tries to impose its value to the other nodes [19], [13], [16], [89]. While these leader-based designs proved effective in local area networks to deploy a secure version of the Network File System [19], it generally cannot scale to large blockchain networks because only one value is decided [15], [84], regardless of the number of proposed values in the system. Recent consensus implementations allowed to commit up to as many proposed values per consensus instance as participating nodes to scale performance [69], [60], [27]. However, some of these variants [69] fail at solving consensus because their binary consensus protocol may not terminate [82]. And the only blockchains that integrate a provably correct consensus implementation that combines block proposals into a *superblock* support simple transactions but cannot execute arbitrary programs or smart contracts [60], [29]. In §IV-C, we will describe the obstacles we overcome to combine multiple proposals of smart contracts creations and invocations during each consensus execution.

III. Goals and Assumptions

We consider an *open permissioned* blockchain model [69], [29] in that a subset of the distributed machines have the permission to run the current instance of the consensus, or to execute smart contracts and transaction requests as well as to maintain the resulting state. This model is called "open" as permission can be revoked and we do not prevent a particular node from obtaining a permission later on: as opposed to Ethereum we simply prevent all nodes from providing the same service at the same time to avoid resource waste (§IV-D4).

We assume *partially synchronous* communication and computation in that the upper bound on the time it takes for a step exists but is unknown [37]. For simplicity, we assume that each permissioned participant runs both a consensus node and a state node and that up to f of these participants (and any of their nodes) can fail arbitrarily by being byzantine. In this case, we call such a participant as a *blockchain node*.

a) The Blockchain problem: We refer to the blockchain problem as the problem of ensuring both the safety and liveness properties that were defined in the literature by Garay et al. [47] and restated more recently by Chan et al. [20], and a classic validity property [29].

Definition 1 (The Blockchain Problem): The *blockchain problem* is to ensure that a distributed set of blockchain nodes maintain a sequence of transaction blocks such that the three following properties hold:

- *Liveness:* if a correct blockchain node receives a transaction, then this transaction will eventually be reliably stored in the block sequence of all correct blockchain nodes.
- *Safety:* the two chains of blocks maintained locally by two correct blockchain nodes are either identical or one is a prefix of the other.
- *Validity:* each block appended to the blockchain of each correct blockchain node is a set of valid transactions (non-conflicting well-formed transactions that are correctly signed by its issuer).

The safety property does not require correct blockchain nodes to share the same copy, simply because one replica may already have received the latest block before another receives it. Note that, as in classic definitions [47], [20], the liveness property does not guarantee that a client transaction is included in the blockchain: if a client sends its transaction request exclusively to byzantine nodes then byzantine nodes may decide to ignore it.

b) Our goal of a secure and efficient middleware for DApps: Our goal is thus to support DApps, by allowing clients (i) to access consistent data despite $f < n/3$ byzantine servers through all sorts of devices and (ii) to serve a large demand generated by network effects as follows:

1) **Lightweight-security:** the users should be able to securely interact with the blockchain from various devices. To access apps, users typically use handheld devices that cannot download blockchain histories due to resource constraints (Ethereum history exceeds 280 GiB [41]). yet they need to interact securely despite unpredictable message delays.
2) **Thousands-TPS:** the volume of transactions that can be served per second should prevent a backlog of requests that grows and leads to congestion. We know that DApps create congestion on Ethereum [30] and EOS [38], and popular applications, like Twitter, exceed 4000 requests per second [64].

Property (1) alleviates the need for clients to download the blockchain history, but requires them to interact securely, which is enabled by limiting the number of failures to f and querying $f+1$ identical copies of the current state to retrieve the correct information as detailed in §IV. Also, we know that to allow users to issue (potentially conflicting) transactions from distinct devices, we need to solve consensus [52]. Property (2) lower bounds the capacity to around 2000 TPS to serve the demand of sharing applications. Although this might be insufficient to run multiple DApps, we explain in §V-G how to shard CollaChain and deploy different DApps to different shards. This would help CollaChain support many DApps smoothly, given its 4500 TPS peak throughput (§V-F).

IV. CollaChain

CollaChain is a collaborative blockchain compatible with the largest ecosystem of DApps, it is optimally resilient against byzantine failures. The layered architecture is depicted in Fig. 1 with a Scalable EVM (SEVM) node at the top and a consensus node at the bottom that can be run on the same machine as a single *blockchain node*. The communication between the consensus node and the SEVM node is event-based and

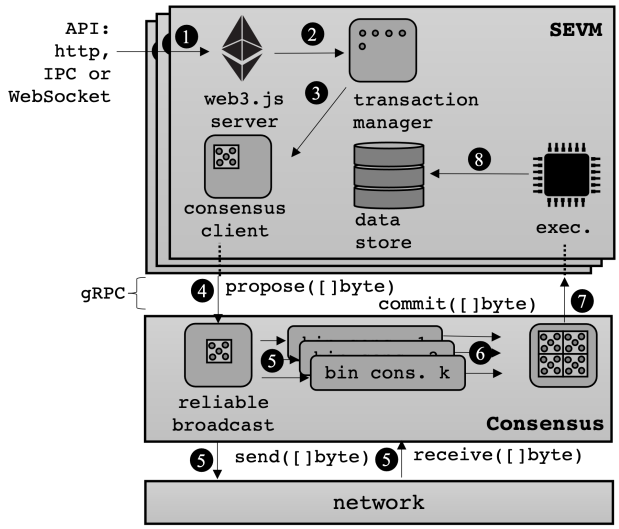

Fig. 1. The architecture of CollaChain. ❶ A client sends a transaction to some replica(s), ❷ at each replica the web3.js server validates transactions and sends them to the transaction manager that ❸ sends a block to the consensus client. ❹ The consensus client proposes it to the consensus protocol. Upon reception of a new block from the consensus client, ❺ the consensus protocol sends it through the network with a reliable broadcast. Remote replicas start participating in the same instance (if not done yet) upon reliably delivering this proposed block. ❻ When the consensus outputs some acceptable blocks, ❼ all of these blocks are combined into a superblock and sent to the SEVM. ❽ As in the EVM, the SEVM is responsible for executing and storing blocks, except that the SEVM will store multiple blocks per consensus instance.

implemented with `gRPC`. Although this presents an execution overhead as both the consensus node and the SEVM node can typically execute on a single machine, this offers greater modularity.

CollaChain offers a secure interface to lightweight clients (§IV-B) and to scale to a geodistributed network of n blockchain nodes, CollaChain reduces the time spent validating transactions as n grows (§IV-C) and increases the amount of blocks committed per consensus instance as n grows (§IV-D). Before discussing these, we present the CollaChain overview through the transaction lifecycle (§IV-A).

A. The transaction lifecyle

In the following, we use the term *transaction* to indistinguishably refer to a simple asset transfer, the upload of a smart contract or the invocation of a smart contract function. The lifecycle of a transaction goes through these subsequent stages:

1. Reception. The client creates a properly signed transaction and sends it to at least one CollaChain node. Once a request containing the signed transaction is received ❶ by the JSON RPC server of the SEVM state machine running within CollaChain, the eager validation (§II) starts. If the validation fails, the transaction is discarded. If the validation succeeds, the transaction is added to a transaction pool. Unlike in Ethereum where the transaction would be propagated to all miners increasing the number of eager validations, CollaChain simply proposes it to the consensus node as follows. If the number of transactions in the pool reaches a threshold, then the transaction manager creates a new proposed block with a number (defined by the threshold) of transactions from the pool ❷. It serializes and sends the proposed block to the consensus client ❸. For the sake of our superblock optimization (cf.§IV-D2) and in contrast with Ethereum, the proposed block does not contain a hash just yet.

2. Consensus. Once the consensus client receives a proposed block, it sends the corresponding byte array to the consensus system by invoking the `propose([]byte)` method ❹. The consensus system starts a new instance of consensus using the new block if it is not currently part of another consensus instance. Otherwise, it adds the new block to the block queue waiting for the current consensus instance to terminate. Like in classic reductions of the general consensus problem to the binary consensus problem [9], [10], CollaChain's consensus execution ❺ consists of an all-to-all reliable broadcast of the blocks among all consensus replicas that trigger as many *binary* consensus instances whose outputs indicate the indices of acceptable blocks ❻ as detailed in §IV-D. The consensus system creates a superblock with all acceptable blocks (§IV-D2) and sends this superblock to the state machine by invoking the `commit([]byte)` method ❼.

3. Commit. When the superblock is received by the `gRPC` server running in the SEVM state machine, the superblock is first deserialized using JSON unmarshalling. The SEVM does the lazy validation (§II) before committing the deserialized transactions ❽. Note that as opposed to the eager validation, all servers execute the lazy validation for a committed transaction, yet it does not prevent CollaChain from scaling to hundreds of nodes (§V-E). Once the superblock is decided, each of its blocks are executed, their hash is included, their results are written to persistent storage on the local disk and the lifecycle ends.

B. Secure interface for lightweight clients

As opposed to classic blockchains, like Ethereum, CollaChain does not require the client[1] interacting with the service to download the blockchain or its block headers. Instead, CollaChain accepts connections from simple javascript-enabled browsers, as can be found on mobile devices [68]. Similar to `geth`, CollaChain supports a `web3.js` API that allows the user to communicate through `http`, IPC or websocket. Given that CollaChain tolerates f failures, it is sufficient for the client to query the same copy of the world state from $f+1$ distinct blockchain servers to guarantee that this copy is consistent. And the client is guaranteed to find this copy at $f+1$ blockchain nodes by contacting $2f+1$ blockchain nodes by assumption. As a result, the client interacts securely with CollaChain without any blockchain records whereas an Ethereum "light" client cannot interact securely due to incomplete blockchain

[1]The term "client" is often used in Ethereum to refer to a node regardless of whether it acts as a server. We use client in the traditional sense of the client-server distinction [81].

records [42]. Hence CollaChain guarantees the *Lightweight-security* property (§III).

C. From the EVM to SEVM

Here we present the modifications we made to the original EVM (and in particular `geth` v1.8.27) in order to obtain the *Scalable EVM*, or *SEVM* for short. More specifically, provided that k transactions are received by CollaChain, we reduce the average number k of transactions each SEVM node eagerly validates to k/n.

1) Reducing the transaction validations: As opposed to each Ethereum server that validates eagerly and lazily each of the k transactions of the system, each of the n CollaChain servers eagerly validates on average k/n transactions. Specifically, only one SEVM node needs to eagerly validates each transaction: the first SEVM node receiving the transaction validates it but does not propagate it to other SEVM nodes but simply proposes it to the consensus. As a result, CollaChain limits the redundant validations, which improves performance. More precisely, if the number of SEVM nodes is n, then each SEVM node does $1 + 1/n$ validations per transaction on average (one lazy validation + $1/n$ eager validation) compared to the two validations needed in `geth`. As n tends to infinity, CollaChain servers validate on average half what `geth` servers validate. In the worst case, where all clients send their transactions to $f + 1 = n/3$ servers simultaneously, then each server will still eagerly validate only $k/3$ transactions.

Note that, as a result of our optimization, a byzantine SEVM node could propose transactions to the consensus without validating them eagerly, in this case two things can happen: i) The transaction is discarded at the lazy validation if invalid ii) The SEVM attempts to execute the invalid transaction, fails at execution and reverses the state to what it was. Either way, there is no impact on the safety of the blockchain. This is also not a DoS vulnerability of CollaChain, as even a byzantine EVM node in Ethereum can propagate invalid transactions to all EVM nodes, forcing all EVM nodes to unnecessarily eager-validate them.

Finally, reducing the validation needed at each SEVM node helps CollaChain reach the *Thousands-TPS* property (§III).

2) Reliably storing superblocks: At each index of the blockchain, our SEVM typically executes many more transactions as part of function `execute_transaction` (lines 1–16) than Ethereum. This is due to the consensus outputting through the `commitChan` channel a superblock containing potentially as many blocks as blockchain nodes (line 3). In Ethereum, blocks are created before the consensus, thus `geth` updates only one block, by setting its state parameters, per consensus instance: `updateBlockState` points a block to its parent, assigns the block header timestamp and the number of transactions associated with a block. This function should thus be invoked for each block before the transactions of the block are executed and persisted, in order to ensure that the data structures are updated properly. To store multiple blocks at the end of the consensus instance, we modified `geth` to `updateBlockState` (line 10) multiple times per consensus instance (one invocation per block) as follows:

More specifically, we reordered the transactions (as disordered transactions could be discarded due to their invalid nonces) and changed the original procedure to guarantee that not only one block but all blocks of our superblock were correctly stored in the transaction and reception tries as a batch of n blocks. Like the C++, python and `geth` software of Ethereum, we reliably store the information in the open source key-value store LevelDB (line 14).

```
1  execute_transaction:
2    // for each superblock received
3    for superblock in node.commitChan do
4      vtxs := ∅ // set of valid transactions
5      for block in superblock do // each block
6        txs := node.txm.deserialize(block) // get txs
7        for tx in txs:
8          if isValid(tx): vtxs.add(tx) // lazy validation
9        // set the corresponding block state and order txs
10       updateBlockState()
11       for tx in vtxs do // for each valid tx...
12         executeTx(tx) // ...execute it
13       done
14       persist(vtxs) // persist valid txs to disk
15     done
16   done
```

3) SEVM support for fast-paced consecutive blocks: Since our consensus system is fast, it creates and delivers superblocks at high frequency through the commit channel to the SEVM. As `geth` does not expect to receive blocks at such a high frequency, it raises an exception outlining that consecutive block timestamps are identical, which never happens in a normal execution of Ethereum. This equality arose because `geth` encodes the timestamp of each block as `uint64`, not leaving enough space for encoding time with sufficient precision. `geth` typically reports an error when consecutive timestamps are identical, due to a strict check that compares the parent block timestamp to the current block timestamp in `go-ethereum/consensus/ethash/consensus.go`: `header.Time < parent.Time`. We changed the original check to `header.Time <= parent`, which allowed for fast-paced executions of consecutive blocks.

4) Bypassing the SEVM resource bottlenecks: After the consensus, the SEVM lazily validates many transactions, updates the memory and storage, which consumes high CPU, memory and IO resources. Typically, high CPU usage slows down the SEVM which results in the increase of the pending list of transactions. Once a threshold of pending transactions are reached, we observe transaction drops. This was evident in our superblock implementation. We observed that consuming each resource one after another, for 10 proposed blocks with a total of 15,000 transactions, would lead to losing transactions requests even on our reasonably-provisioned AWS instances featuring 16 GB RAM and 4 vCPUs (Fig. 4). This is why we made SEVM fully process one proposed block of the superblock at a time allowing it to alternate frequently between CPU-intensive (verifying signatures and transaction executions) and memory-intensive (state trie write) and IO-intensive (reception/transaction tries writes) tasks. Thanks to

this optimized implementation of the superblock, SEVM does not experience bottlenecks as the number of nodes increases (cf. §V-E).

D. A BFT Consensus for SEVM

As opposed to smart contract blockchains that decide (at most) one of the proposed blocks, CollaChain decides a superblock that results from its consensus system combining multiple proposed blocks into a single decision. In the ideal case, agreeing on a superblock thus allows to commit $\Omega(n)$ blocks of distinct transactions at the end of a single consensus instance.

1) Peer-to-peer network: The peer-to-peer (P2P) network of the consensus system is implemented using golang's RPC package `net/rpc`. The consensus node reads consensus network configuration from a `yaml` configuration file upon initialization. The configuration file contains the network size n, a port number p and a list of socket addresses specified in `ip:port` format. The consensus node sets up a gRPC server on port p for consensus messages. The list of socket addresses are gRPC endpoints of consensus nodes. To prevent byzantine nodes from eavesdropping, all communications use TLS. We show that the overhead induced by the encryption layer of TLS is negligible in Fig.5 of §V.

2) Increasing the decision size: The requirement of deciding at most one block is too restrictive to scale with the number n of consensus participants. Whatever n is, the consensus decides at most one single block. As our goal is to scale with the number n of consensus participants, we allow CollaChain to decide a combination of all the $\Omega(n)$ proposed blocks to make a superblock (line 30). This helps ensuring the *Thousand-TPS* property (§II) as we explained before. Note that the same optimization was shown effective for Red Belly Blockchain [28] to scale to hundreds of consensus participants, but Red Belly only supports the Bitcoin scripting language and not smart contracts. The drawback of this superblock is that its size increases with the number n of participants, and so does its propagation time. To cope with arbitrary delays, we build our consensus upon DBFT [27] that is partially synchronous [37] and was recently proved correct via model checking [82], [12]. More specifically, this consensus protocol remains safe whatever delay it takes to deliver a message and when messages are delivered in a bounded (but unknown) amount of time the consensus protocol terminates.

```
20 index := 0 // consensus instance
21 blockQueue := ∅ // pending block to propose
22 commitChan := chan []byte  // commit channel
23
24 start_new_consensus():
25   index := index + 1 // increment round
26   myBlock := blockQueue.peek() // get block proposal
27   superblock := consensus_propose(myBlock) // block
28   if (myBlock is in superblock) then
29     blockQueue.poll() // dequeue proposal
30   commitChan := superblock // send to gRPC srvce
```

3) The consensus protocol: The protocol is divided in two procedures, `start_new_consensus` at lines 24–30 that spawns a new instance of (multivalue) consensus by incrementing the replicated state machine `index`, and `consensus_propose` at lines 37–51 that ensures that the consensus participants find an agreement on a superblock comprising all the proposed blocks that are acceptable. The idea of `consensus_propose` builds upon classic reduction [9], [10] by executing an all-to-all reliable broadcast [14] to exchange n proposals, guaranteeing that any block delivered to a correct process is delivered to all the correct processes: any delivered proposal is stored in an array *proposals* at the index corresponding to the identifier of the broadcaster. The main difference is that these reductions use a probabilistic binary consensus algorithm while our binary consensus is deterministic.

A binary consensus at index k is started with input value `true` for each index k where a block proposal has been recorded (line 35). To limit errors, CollaChain uses the formally verified deterministic binary consensus of DBFT [27], we omit the pseudocode for the sake of space and refer the reader to the formal verification of the protocol [82], [12]. As soon as some of these binary consensus instances return 1, the protocol spawns binary consensus instances with proposed value false for each of the non reliably delivered blocks at line 44. Note that this invocation is non-blocking. As the reliable broadcast fills the `block` in parallel, it is likely that the blocks reliably broadcast by correct processes have been reliably delivered resulting in as many invocations of the binary consensus with value `true` instead. Once all the n binary consensus instances have terminated, i.e., `decidedCount == n` at line 46, the superblock is generated with all the reliably delivered blocks for which the corresponding binary consensus returned `true` (lines 47–51). At the end of `start_new_consensus`, if the superblock of the consensus contains the block proposed, then this block is removed from the `blockQueue` at lines 28 and 29 to avoid reproposing it later.

```
31 blocks := ∅ // blocks delivered by reliable bcast
32
33 upon reliable_broadcast.deliver(i, block):
34     blocks[i] := block // append block to list
35     decBlocks[i] := b_consensus.propose(i, true)
36
37 consensus_propose(myBlock):
38   decCount := 0 // # decided bin. cons. instances
39   decBlocks :=  ∅
40   reliable_broadcast.broadcast(myId, myBlock)
41   wait until ∃i : b_consensus.decide(i) == true
42      for j  from 0 to n do
43        if blocks[j] is null then
44          decBlocks[j] := b_consensus.propose(j, false)
45      decCount := decCount+1
46    wait until decCount == n
47      superblock := ∅
48      for i from 0 to n do
49        if decBlocks[i] is true then
50          superblock.add(blocks[i])
51   return superblock
```

4) Proof-of-stake and membership change: As mentioned in §III, CollaChain is an open blockchain and changes its membership (SEVM and consensus nodes) at runtime with a reconfiguring smart contract (provided in Appendix A). To cope with bribery attacks, CollaChain relies on the proof-of-stake (PoS) design common to other blockchains [50],

[43] that assumes that users who have stake are more likely to behave correctly. Initially, the blockchain is setup with a membership smart contract that accepts a `rotate` method that outputs a random sample of n consensus participants among all potential blockchain participants with a preference for participants with the most assets, similar to a sortition [50]. Initially and periodically, the correct participants invoke the `rotate` function that outputs new consensus participants for the subsequent blocks. Traditional SSL authentication guarantees that the byzantine participants are ignored. As in Eth2 [43], neither does the system start nor does the `rotate` method is invoked until sufficiently many participants exist. To incentivize participants, CollaChain can reward consensus participants just like Bitcoin's miners [71], however, this reward has not been implemented.

E. Proofs of correctness

In this section, we show that CollaChain solves the blockchain problem (Def. 1). Note that the proofs that CollaChain also guarantees *Lightweigth-Security* and *Thousands-TPS* (§III) follows directly from the protocols (§IV-B and §IV-C1) and the experimental results (§V). For the sake of simplicity in the proofs, we assume that there are as many nodes playing the roles of consensus nodes and state nodes, and one state node and one consensus node are collocated on the same physical machine.

Lemma 1: If at least one correct node consensus-proposes to a consensus instance i, then every correct node decides on the same superblock at consensus instance i.

Proof. If a correct node p consensus-proposes, say v, to a consensus instance i, then p reliably broadcast v at line 40. By the reliable broadcast properties [14], we know that v is delivered at line 33 at all correct nodes. By assumption, there are at least $2f + 1$ correct proposers invoking the reliable broadcast, hence all correct proposers eventually populate their block array with at least one common value. All correct proposers will thus have input true for the corresponding binary consensus instance at line 35. Now it could be the case that other values are reliably-broadcast by byzantine nodes, however, reliable broadcast guarantees that if a correct proposer delivers a valid value v, then all correct proposers deliver v. By the validity and termination properties of the DBFT binary consensus [27], the decided value for the binary consensus instance at line 41 is the same at all correct nodes. It follows that all correct nodes have the same bit array of decBlocks values at line 49 and that they all return the same superblock at line 51 for consensus instance i. □

The next three theorems show that CollaChain satisfies each of the three properties of the blockchain problem (Definition 1).

Theorem 1: CollaChain satisfies the safety property.

Proof. The proof follows from the fact that any block B_ℓ at index ℓ of the chain is identical for all correct blockchain nodes due to Lemma 1. Due to network asynchrony, it could be that a correct node p_1 is aware of block $B_{\ell+1}$ at index $\ell+1$, whereas another correct node p_2 has not created this block $B_{\ell+1}$ yet. At this time, p_2 maintains a chain of blocks that is a prefix of the chain maintained by p_1. And more generally, the two chains of blocks maintained locally by two correct blockchain nodes are either identical or one is a prefix of the other. □

Theorem 2: CollaChain satisfies the validity property.

Proof. By examination of the code at line 8, only valid transactions are executed and persisted to disk at every correct node. It follows that for all indices ℓ, the block B_ℓ is valid. □

Theorem 3: CollaChain satisfies the liveness property.

Proof. As long as a correct replica receives a transaction, we know that the transaction is eventually proposed by line 27. The proof follows from the termination of the consensus algorithm [12] and the fact that CollaChain keeps spawning new consensus instances as long as correct replicas have pending transactions. □

V. Evaluation of CollaChain

In this section, we present the experimental evaluation of CollaChain, compare it against other blockchains (§V-B), evaluate it when running across different continents (§V-E) and with a Twitter DApp (§V-F).

A. Experimental setup

We use up to 200 AWS virtual machines from 10 regions located in separate countries across 5 continents: Ohio, Mumbai, Seoul, Singapore, Sydney, Tokyo, Canada, Frankfurt, London, Paris, Stockholm, São Paulo. All machines run Ubuntu v18.04.3 LTS, golang v1.13.1. When not specified otherwise, the experiments consist of having clients sending 1500 distinct transactions to each SEVM nodes that exchange with their respective consensus node to spawn a consensus instance. The client machines are of type c5.xlarge with 4 vCPUs and 8 GiB of memory, the SEVM nodes are of type c5.2xlarge with 8 vCPUs, 16 GiB of memory, and the consensus nodes are of type c5.4xlarge with 16 vCPUs, 32 GiB of memory.

As CollaChain is compatible with Ethereum, we reuse libraries of the JavaScript runtime environment `Node.js`: we create wallet addresses with `ethereumjs-wallet`, pre-sign transactions to transfer assets, upload or invoke a smart contract with `ethereumjs-tx`, and serialize these transactions before saving them to a JSON file. The client iterates through the serialized JSON file and sends the transactions to the SEVM using `web3.eth.sendSignedTransaction` through the `web3.js` javascript API using `http`. Hence, this offloads the encryption time from the performance measurement. All presented data points are averaged over at least 3 runs.

B. Comparison with other blockchains

Here we compare the performance of CollaChain to Quorum [22] from JP Morgan/Consensys and Concord [83] from VMware that both support Ethereum smart contracts. While evaluating all blockchains is out of the scope of this paper, note that Table I provides a comparison of CollaChain to non-sharded blockchains while sharded blockchains are discussed in §V-G and §VI. We have also explored Burrow [61] that is

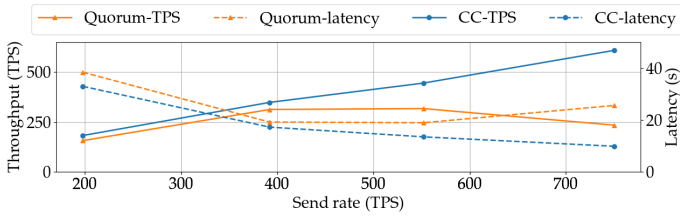

Fig. 2. Comparison of throughput and latency between CollaChain (CC) and Quorum against sending rate.

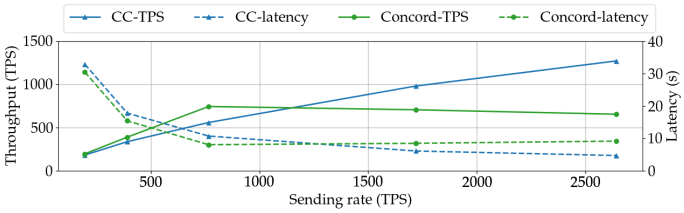

Fig. 3. Comparison of throughput and latency between CollaChain (CC) and Concord against sending rate.

unstable [78] and Ethermint whose open issues [2] prevented us from evaluating it.

Figure 2 reports the latency and throughput of both CollaChain and Quorum. CollaChain outperforms Quorum both in terms of latency and throughput because Quorum does not use superblocks. Interestingly, we also observe that the performance of Quorum starts decreasing as the sending rate increases whereas the performance of CollaChain keeps increasing, this is seemingly due to the growing backlog of requests in Quorum that induces congestion. Unfortunately, we could not test a sending rate of 800 TPS and above as Quorum would start losing requests, which confirms previous observations [78].

Figure 3 compares the throughput and latency of Concord and CollaChain. As Concord suffers from known configuration issues [24] that prevented us from running it on a distributed system, we ran both Concord and CollaChain on a single c5.9xlarge machine with 4 client machines. Concord slightly outperforms CollaChain with low sending rates, however, as the sending rate increases, CollaChain outperforms Concord significantly.

C. Effect of validation reduction using the SmartBank DApp

In order to assess the impact of the validation optimization (§IV-C) of the SEVM on the performance, we measured the time spent validating eagerly when running the smart bank DApp that is part of BlockBench [34]. To this end, we instrumented its `writeCheck` function to measure both the total time Δ^n_{SEVM} spent treating k calls and the average time δ^n_{SEVM} spent by each server of SEVM validating eagerly these calls on n nodes, to deduce the rest of the treatment time not affected by the validation optimization $\beta = \Delta^n_{SEVM} - \delta^n_{SEVM}$.

Based on this measurement, we could deduce the time δ_{EVM} the EVM would spend validating eagerly without the validation optimization: $\delta_{EVM} = n \cdot \delta^n_{SEVM}$. In particular, regardless of n, we know that the EVM would spend $\Delta_{EVM} = \beta + \delta_{EVM}$ to treat the function calls. By contrast, depending on n, the SEVM would spend $\Delta^n_{SEVM} = \beta + \delta^n_{SEVM}$. As $\delta^n_{SEVM} = \delta_{EVM}/n$, we know that $\lim_{n\to\infty}(\delta^n_{SEVM}) = 0$. This means that, with n servers, the EVM slowdown compared to the SEVM is:

$$S = \frac{\Delta_{EVM} - \Delta^n_{SEVM}}{\Delta^n_{SEVM}} = \frac{\delta_{EVM} + \delta^n_{SEVM}}{\beta + \delta^n_{SEVM}}.$$

As n tends to infinity, we thus have a slowdown of:

$$\lim_{n\to+\infty} S = \frac{\delta_{EVM}}{\beta}.$$

Our measurement obtained with $k = 6000$ transactions and $n = 4$ revealed that $\delta^n_{SEVM} = 0.61$ seconds and $\Delta^n_{SEVM} = 5.66$ seconds. Hence, we have $\beta = 5.66 - 0.61 = 5.05$. As $n = 4$, we have $\delta_{EVM} = 4 \times 0.61 = 2.44$ so that $\Delta_{EVM} = 5.05 + 2.44 = 7.49$. This means that the EVM would take $S = 32\%$ more time than the SEVM to treat these DApp requests. Finally, as n tends to infinity, the slowdown of the EVM over the SEVM would become 48%.

D. Storing the superblock efficiently

To measure the impact on performance of storing each block separately, we implemented a naive method that stores the whole block at once. More precisely, we changed the storing loop (§IV-C2) by simply removing the inner `for` loop at lines 5–15 that persisted one (sub-)block at a time, in order to persist the superblock once and for all. In this experiment, we setup a network of 2 clients machines, 10 consensus machines and 10 SEVM machines where clients send 1500 distinct transactions to each EVM node for a total of 15000 transactions.

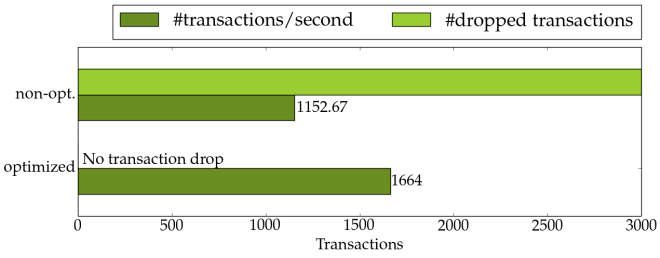

Fig. 4. Performance difference when processing each block of a superblock at a time (optimized) and when processing the entire superblock at once (non-optimized)

Figure 4 compares the performance obtained with CollaChain (superblock optimized) and with the naive approach (superblock non-optimized). The throughput of CollaChain (superblock optimized) is 44% higher than the throughput of the naive approach. This is because trying to persist a large superblock that comprises 10 blocks leads to I/O congestion. One might argue that multi-threading block writes and transaction executions could also solve this issue. However, this is not possible as the execution and writing of blocks should happen sequentially. Interestingly, In addition, we observed that 3000 transactions get dropped, which represents 20% of all transactions, when executing the naive approach. This is due to CPU overload: executing a superblock of 10 blocks within a single loop

iteration is more CPU intensive than executing one block per iteration because between two block executions the CPU resource can be allocated to other tasks. If the clients keep sending transactions while the CPU usage of the SEVM node reaches 100%, the SEVM starts dropping incoming transactions as soon as it cannot hold any more transactions. These results show the importance of optimizing the superblock storage for CollaChain to not suffer transaction drops.

E. World-wide scalability

To evaluate the scalability of the performance of CollaChain, we deployed CollaChain in 10 regions spanning 5 continents: Canada, London, Mumbai, Oregon, Paris, São Paulo, Singapore, Stockholm, Sydney and Tokyo. As previously mentioned, we consider for simplicity that each participant is running both a consensus node and an SEVM node so that we can consider each participant as a single entity, out of all of which at most a third can be faulty.

Figure 5 depicts the throughput without end-to-end ecryption (w/o TLS) and with encryption (with TLS) of CollaChain as we run CollaChain on more and more machines: We start our experiment with 20 machines spread evenly in the 10 countries and add machines by group of 20 evenly spread in the 10 countries until we reach 200 machines. We observe that the throughput increases as we increase the number of nodes from 1100 TPS at 20 machines to 2038 TPS at 200 machines, demonstrating the scalability of CollaChain even in a geo-distributed setting. The curve flattens out at large scale between 140 and 200 nodes, indicating that the gain obtained in throughput by adding more machine becomes lower and lower. This is due to the numerous machines consuming the available bandwidth. Finally, we observe, as expected, that the TLS encryption comes at a cost, however, this overhead is negligible in comparison of the overall performance as the peak throughput with TLS (1960 TPS) is only 4% lower than the peak throughput without TLS (2038 TPS).

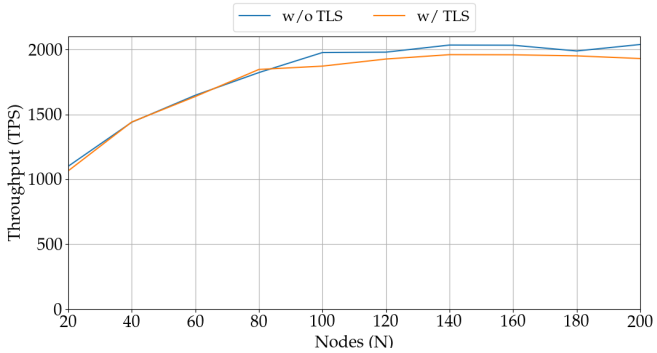

Fig. 5. Throughput of CollaChain when deployed in a geo-distributed AWS environment of 10 countries across 5 continents with and without TLS

Figure 6 shows the latency of transactions of CollaChain in the aforementioned geo-distributed environment as the number of nodes increases. We can observe that the latency increases with the number of nodes. We observe similar minimum latencies across all system sizes but the 99^{th} percentile indicates that some requests can take much longer especially at large scale: the transactions take less than 10 seconds to execute on up to 40 nodes while they take less than 40 seconds to execute at 200 nodes. It is important to note that these latencies can be viewed as time for a transaction to become final: thanks to our deterministic byzantine fault tolerance consensus (§IV-D), transactions are committed (and thus final) as soon as the consensus ends and the superblock is selected. This differs from classic blockchains [88], [55] whose consensus is reached after the block is appended and after more "block confirmations" occur. Interestingly, this increasing latencies do not prevent the throughput from scaling with the number of machines as we discussed earlier (Fig. 5). This is precisely due to the superblock optimization: As more machines participate, more blocks get proposed and running consensus takes more time, which increases the latency, however, the number of transactions decided per consensus instance also increases, which guarantees scalability.

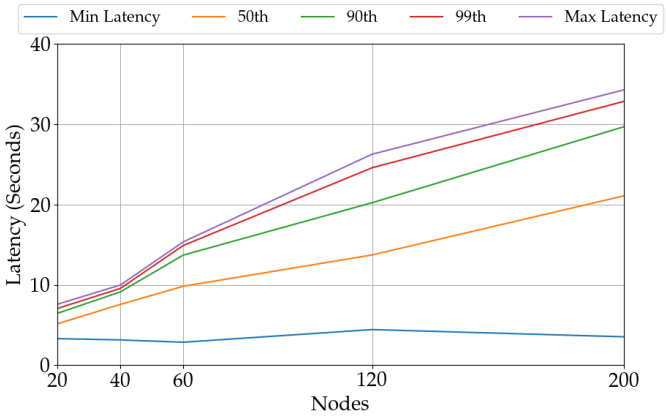

Fig. 6. Latency of CollaChain when deployed in a geodistributed AWS environment of 10 countries across 5 continents

F. Twitter DApp evaluation

To evaluate how fast CollaChain can treat smart contract invocations under a realistic workloads, we ran the Twitter DApp of the DIABLO framework [11] on top of 4 consensus nodes and 4 SEVM nodes and report on the performance as time elaspses. DIABLO is a benchmark suite for blockchains that features DApps written in different smart contract programming languages. It features a Twitter DApp written in Solidity whose smart contract sends 140-character messages following a burst workload experienced during the release of the *Castle in the Sky* anime.

Figure 7 depicts the performance results obtained while running this Twitter DApp on top of CollaChain. To achieve the high burst Twitter workload of 143,000 TPS, we had to deploy as many clients as SEVM nodes. We can observe that CollaChain handles the load burst of the Twitter workload by committing up to 4509 transactions per second. During the same experiments, we observed that a latency with 50^{th}, 90^{th}, 99^{th} percentiles as 5.367, 8.171 and 195.337 seconds. Despite this high workload burst, CollaChain continues executing properly,

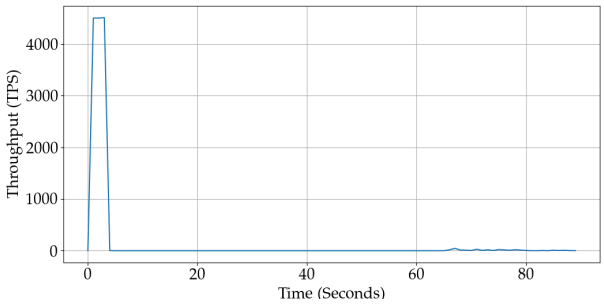

Fig. 7. Twitter DApp throughput when running on top of CollaChain

committing other transactions at a slower rate with a minor peak around 67 seconds. This demonstrates that CollaChain can deliver high throughput and handles workload bursts as can be found in realistic applications.

G. Linear speedup with sharding

As we explain in the related work, sharding proved instrumental in boosting performance of blockchains. The idea of sharding stems from distributed database research where a database table gets split into sub-tables, each sub-table is stored on a partition of machines called a *shard*. When a request on a entry of the table is issued, the shard managing this entry is responsible for handling this request. Hence, requests on distinct entries can execute in parallel on distinct shards, allowing performance to increase (ideally linearly) with the number of shards.

In blockchain, there exists various ways of implementing sharding. With traditional blockchain sharding, each shard is responsible of a subset of the transactions and runs an independent consensus instance to agree upon the ordering of these transactions. This is the approach taken by Dfinity [55], Elastico [66], RapidChain [90] and Omniledger [59]. With verification sharding, every shard participates in the same consensus instance and stores the global state, however, each shard is responsible of verifying a different set of transactions. This is the approach taken by Red Belly Blockchain [29].

We implemented sharding on top of CollaChain using a traditional blockchain sharding. Shards can be spawned on demand by a dedicated built-in smart contract, hence resulting in a beacon chain with shard chains structure similar to the upcoming Ethereum 2 [43]. Participants can deposit some assets on the beacon chain in order to spawn a new shard, which creates a new blockchain instance where the participants have an account with a balance corresponding to the assets they deposited on the beacon chain—this typically allows participants to transact within the shard without having to transact on the beacon chain.

Table II presents the performance obtained on up to 32 small machines with 2 vCPU and 8 GiB memory running Ubuntu 20.04 when we increase the number of shards. The beacon chain (1 shard) delivers 470.79 TPS but when coupled with two other shards (3 shards) it delivers 2.98× higher performance,

#Shards	1	2	3
Throughput	470.79 TPS	951.54 TPS	1405.42 TPS

TABLE II
COLLACHAIN PERFORMANCE SCALES ALMOST LINEARLY WITH THE NUMBER OF SHARDS

which demonstrates a speedup very close to linear. This is no suprise given that each shard runs on separate machines and use distinct resources. Although sharding makes the implementation of cross-shard transactions quite complex, we could offer cross-shard transactions by decoupling these transactions into separate withdrawals an credits, as was previously suggested in Prism [85], for applications where the atomicity of transactions is not a requirement.

VI. RELATED WORK

To decentralize the computation from large data stores [32], [21], [26], various work focused on user/edge-centric computing [48]. Solid [67] distributes private data into pods whose user manages permissions. Lightweight middleware [58] exploit WebRTC to avoid downloading a blockchain. These solutions do not offer the execution transparency of blockchains, which is key to prevent user manipulations [75].

a) Payment blockchains: Some blockchains are designed for high transactions throughput at large scale, but were not designed to support DApps [50], [28], [80], [65], [54]. This is the case of ResilientDB [54] that exploits topology-awareness to parallelize consensus executions, the Red Belly Blockchain [28] that shares our superblock optimization or Mir [80] that deduplicates transaction verifications. Stellar [65] is an in-production blockchain running in a geodistributed setting while Algorand [50] introduced the sortition our membership change builds upon. Although progresses are being made towards smart contract support, these blockchains do not run DApps.

b) Fast smart contract executions: Solana [88] builds upon Proof-of-History (PoH) to reduce message overhead. Solana provides high performance by offering optimistic consensus, hoping that a single block gets notarized at each index, and thanks to the vertical scaling of its validator nodes: validator nodes feature 1TB SSD disk and 2 Nvidia V100 GPUs for benchmarking [1] and 128 GiB memory is required [4], which is twice as much as the most powerful machines we used in our experiments.

c) Towards byzantine fault tolerant blockchains: Upper-bounding the number f of byzantine failures allow to solve consensus to avoid forks. Ethereum comes with proof-of-work and proof-of-authority (PoA) in the two mainstream Ethereum programs, called `parity` and `geth`. The idea of proof-of-authority is to have a set of n permissioned validators, among which f can be malicious or *byzantine* [72], that generate new blocks [44], [73], [56]. Unfortunately, both proof-of-authority protocols in `parity` and `geth` have recently been shown vulnerable to the attack of the clone when messages take longer than expected [39].

d) Tolerating unpredictable bounded delays: To cope with unpredictable message delays, blockchains cannot rely on synchrony. Cosmos [18], sometimes referred to as the Internet of Blockchain, is a network of interoperable blockchains that builds upon the Tendermint state machine replication [15]. Ethermint [45] is a blockchain that combines the partially synchronous Tendermint consensus protocol [16] with the EVM. Ethermint is still under active development [3] and we could not benchmark it. In particular, we found some issues that prevented us from deploying it like a nonce management limitation, which resulted in rejecting consecutive transactions sent in a short period of time [2]. Other researchers who managed to deploy an older version of Ethermint, reported a peak throughput of 100 TPS obtained with a single validator node [33], however, Tendermint reached 438 TPS [18].

Zilliqa [91] is a blockchain that supports smart contracts and reaches consensus with PBFT [19]. We are not aware of any performance evaluation of Zilliqa but its state machine, Scilla, executes non Turing complete programs but slower than the EVM when the state size increases [77]. Therefore, it is unlikely that it would yield higher throughputs than our CollaChain for large state sizes. Chainspace [5] introduced a distributed atomic commit protocol termed S-BAC for smart contract transactions. Coupled with the BFT-SMaRt [13] consensus protocol, Chainspace can support trustless use of DApps. However, it has only been able to achieve up to 350 TPS, offering a limited support for DApps.

e) Evaluations of BFT blockchains: Quorum [22] is a blockchain that supports Ethereum smart contracts and reaches consensus with the Istanbul Byzantine Fault Tolerant (IBFT) consensus algorithm. Just like CollaChain, the byzantine fault tolerance of Quorum makes it well-suited for mobile devices to interact wth DApps securely without downloading the blockchain. Moreover, it seems that few optimizations could help it treat a large number of transactions per second [7]. Unfortunately, Quorum loses requests (§V-B).

SBFT [53] is a byzantine fault tolerant consensus algorithm that exploits threshold signatures to reduce the communication complexity of PBFT but commits, like PBFT, at most one proposed block per consensus instance. It was shown to reach consensus on 378 smart contract requests per second when deployed within one continent and 172 requests per second across multiple continents. Concord [83] is a blockchain that combines a lightweight C++ implementation of the EVM with SBFT, however, its publicly available version has open issues [24] that prevent it from being deployed on distinct physical machines but we showed that Concord, although slower than CollaChain, reached the encouraging throughput of 1000 TPS on 4 nodes within the same physical machine. It could be the case that future versions will scale.

f) Sharding: As we presented in §V-G, one can multiply the performance of a blockchain, including CollaChain, by adding more shards. Dfinity [55] coined as the Internet Computer is an open permissioned blockchain. Dfinity scales horizontally thanks to its committees, with an assumed majority of correct members, that act like shards. It achieves high block production throughput thanks to concurrent execution of cannisters, isolated pieces of code compiled to WASM that act as smart contracts and offer low latency to read requests. The difference with CollaChain, is that a block produced is not necessarily final: a verifiable random function is used to rank block proposers and if an adversarial one is rank highest, it could propose conflicting blocks that are notarized, hence leading to a fork. Additional assumptions are needed for the nodes to agree on the chain with the highest block weight. CollaChain solves consensus before appending blocks.

The `move` approach [46] moves accounts and computation from one smart contract enabled blockchain to another. The smart contract of the first blockchain is locked before any participant creates it in the second blockchain. This allows to scale the throughput of the congested DApp CryptoKitties with the number of shards. Eth2 [43] relies on a beacon chain and will feature 64 shard chains to improve the scalability of Ethereum. The uniqueness of the beacon chain guarantees a consistent view of current state but cannot handle accounts and smart contracts. The validators of the shard chain do not need to download and run data for the entire network.

PRISM [86] is a proof-of-work blockchain that shards the blockchain into m voter chains and exploits three types of blocks in a block tree. The voter blocks are used to vote for proposer blocks grouped per level in the block tree. Once a proposer block is elected, transaction blocks that are pointed to by the proposer block are committed. Prism peaks at 19K TPS by ignoring the eager validation completely, which exposes it to DoS attacks. To ensure the copy of the blockchain state is not corrupted, a user needs first to download the block headers, a time- and space-consuming task ill-suited for running DApps on handheld devices.

VII. CONCLUSION

CollaChain is a collaborative blockchain compatible with the largest ecosystem of DApps that treats thousands of requests per second and scales to hundreds of machines world-wide. It builds upon recent advances in deterministic byzantine fault tolerance (BFT) consensus algorithms to avoid forks and offers finality without having to wait for block confirmations. Its key novelties lie in (i) having smart contract execution nodes collaborating to minimize validations and in (ii) having consensus nodes collaborating to combine their block proposals into a committed superblock of smart contracts.

Our experiments demonstrate that CollaChain is an appealing BFT middleware for individuals to exchange in a fully distributed fashion. We showed that one instance of CollaChain handles a peak throughput of 4500 TPS under a Twitter DApp. We also showed how to interconnect different shard instances of CollaChain to scale almost linearly, to support potentially as many DApps as shards. This, combined with the ability of one instance (or shard) to scale to hundreds of nodes spread in 5 continents makes CollaChain an appealing BFT middleware for DApp services. This new model departs from the centralization trend of the sharing economy services to offer more transparent and fault tolerant services to individuals.

References

[1] Performance metrics, 2021. Accessed: 2021-12-09, https://docs.solana.com/cluster/performance-metrics.

[2] Problem: reverted contract deployment transaction don't increase sender's nonce, 2021. Accessed: 2021-12-09, https://github.com/tharsis/ethermint/issues/808.

[3] tharsis/ethermint, 2021. Accessed: 2021-12-09, https://github.com/tharsis/ethermint.

[4] Validator requirements, 2021. Accessed: 2021-12-09, https://docs.solana.com/running-validator/validator-reqs.

[5] Mustafa Al-Bassam, Alberto Sonnino, Shehar Bano, Dave Hrycyszyn, and George Danezis. Chainspace: A sharded smart contracts platform, 2017.

[6] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolić, Sharon Weed Cocco, and Jason Yellick. Hyperledger fabric: A distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, pages 30:1–30:15, 2018.

[7] Arati Baliga, I Subhod, Pandurang Kamat, and Siddhartha Chatterjee. Performance evaluation of the quorum blockchain platform. Technical Report 1809.03421, arXiv, 2018.

[8] Shehar Bano, Mathieu Baudet, Avery Ching, Andrey Chursin, George Danezis, Francois Garillot, Zekun Li, Dahlia Malkhi, Oded Naor, Dmitri Perelman, and Alberto Sonnino. State machine replication in the libra blockchain, 2019. Accessed: 2019-10-01, https://developers.libra.org/docs/assets/papers/libra-consensus-state-machine-replication-in-the-libra-blockchain.pdf.

[9] Michael Ben-Or, Ran Canetti, and Oded Goldreich. Asynchronous secure computation. In *ACM STOC*, pages 52–61, 1993.

[10] Michael Ben-Or, Boaz Kelmer, and Tal Rabin. Asynchronous secure computations with optimal resilience (extended abstract). In *ACM PODC*, pages 183–192, 1994.

[11] Harold Benoit, Vincent Gramoli, Rachid Guerraoui, and Christopher Natoli. Diablo: A distributed analytical blockchain benchmark framework focusing on real-world workloads. Technical Report 285731, EPFL, 2021.

[12] Nathalie Bertrand, Vincent Gramoli, Igor Konnov, Marijana Lazic, Pierre Tholoniat, and Josef Widder. Compositional verification of byzantine consensus. Technical Report hal-03158911, HAL, June 2021.

[13] Alysson Bessani, Joao Sousa, and Eduardo E. P. Alchieri. State machine replication for the masses with BFT-SMaRt. In *Proceedings of the 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 355–362, 2014.

[14] Gabriel Bracha. Asynchronous Byzantine agreement protocols. *Inf. Comput.*, 75(2):130–143, 1987.

[15] Ethan Buchman. Tendermint: Byzantine fault tolerance in the age of blockchains, 2016. MS Thesis.

[16] Ethan Buchman, Jae Kwon, and Zarko Milosevic. The latest gossip on BFT consensus. Technical Report 1807.04938v3, arxiv, 2018.

[17] Vitalik Buterin. Ethereum whitepaper, 2020. Accessed: 2020-11-14, https://ethereum.org/en/whitepaper/.

[18] Daniel Cason, Enrique Fynn, Nenad Milosevic, Zarko Milosevic, Ethan Buchman, and Fernando Pedone. The design, architecture and performance of the tendermint blockchain network. In *2021 40th International Symposium on Reliable Distributed Systems (SRDS)*, pages 23–33, 2021.

[19] Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, 20(4):398–461, November 2002.

[20] Benjamin Y. Chan and Elaine Shi. Streamlet: Textbook streamlined blockchains. In *Proceedings of the 2nd ACM Conference on Advances in Financial Technologies*, pages 1–11, 2020.

[21] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2), June 2008.

[22] JPMorgan Chase. Quorum whitepaper. Accessed: 2020-12-04, https://github.com/ConsenSys/quorum/blob/master/docs/Quorum%20Whitepaper%20v0.2.pdf.

[23] Huan Chen and Yijie Wang. SSChain: A full sharding protocol for public blockchain without data migration overhead. *Pervasive and Mobile Computing*, 59:101055, 2019.

[24] Error compiling conc_genconfig.cpp tool undefined reference to log4cplus and boost, May 2020. Accessed: 2020-11-28, https://github.com/vmware/concord/issues/41.

[25] Price Waterhouse Coopers. Sharing or paring? growth of the sharing economy, 2015.

[26] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally distributed database. *ACM Trans. Comput. Syst.*, 31(3), 2013.

[27] Tyler Crain, Vincent Gramoli, Mikel Larrea, and Michel Raynal. DBFT: efficient leaderless Byzantine consensus and its application to blockchains. In *Proc. 17th IEEE Int. Symp. Netw. Comp. and Appl (NCA)*, pages 1–8, 2018.

[28] Tyler Crain, Christopher Natoli, and Vincent Gramoli. Evaluating the Red Belly Blockchain. Technical Report 1812.11747, arXiv, 2018.

[29] Tyler Crain, Christopher Natoli, and Vincent Gramoli. Red belly: a secure, fair and scalable open blockchain. In *IEEE S&P*, pages 1501–1518, May 2021.

[30] Cryptokitties craze slows down transactions on ethereum, Dec. 2017. Accessed: 2020-11-14.

[31] DappRadar. DappRadar 2020 Q3 Dapp industry report, Oct. 2020. Accessed: 2020-11-14.

[32] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 205–220, 2007.

[33] Omar Dib, Kei-Leo Brousmiche, Antoine Durand, Eric Thea, and Elyes Ben Hamida. Consortium blockchains: Overview, applications and challenges. *International Journal on Advances in Telecommunications*, 11:51–64, 2018.

[34] Tien Tuan Anh Dinh, Ji Wang, Gang Chen, Rui Liu, Beng Chin Ooi, and Kian-Lee Tan. Blockbench: A framework for analyzing private blockchains. In *SIGMOD*, pages 1085–1100, 2017.

[35] Drife. Taxi 3.0 ride-hailing reimagined - whitepaper. Accessed: 2020-11-28, https://www.drife.one/docs/DRIFE-WhitePaper.pdf.

[36] D.tube - turning the tables in the social media industry, june 2019. https://token.d.tube/whitepaper.pdf.

[37] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):pp.288–323, 1988.

[38] EOS enters congestion mode due to EIDOS airdrop. Accessed: 2020-12-07, https://blog.coinbase.com/eos-enters-congestion-mode-due-to-eidos-airdrop-3d3f82081074.

[39] Parinya Ekparinya, Vincent Gramoli, and Guillaume Jourjon. The Attack of the Clones against Proof-of-Authority. In *Proceedings of the Network and Distributed Systems Security Symposium (NDSS'20)*, Feb 2020.

[40] Accessed: 2020-12-07, https://ethereum.org/en/developers/docs/nodes-and-clients/.

[41] Accessed: 2020-12-07, https://blog.ethereum.org/2019/07/10/geth-v1-9-0/.

[42] Accessed: 2020-12-07, https://docs.ethhub.io/using-ethereum/ethereum-clients/geth/#:~:text=Minimum%3A,space%20to%20sync%20the%20Mainnet.

[43] The eth2 upgrades. Accessed: 2020-11-14, https://ethereum.org/en/eth2/.

[44] Ethereum Proof-of-Authority Consortium - Azure. https://docs.microsoft.com/en-us/azure/blockchain/templates/ethereum-poa-deployment.

[45] Ethermint. Accessed: 2020-11-14, https://github.com/cosmos/ethermint.

[46] E. Fynn, A. Bessani, and F. Pedone. Smart contracts on the move. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 233–244, 2020.

[47] Juan A. Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol: Analysis and applications. In *34th Annu. Int. Conf. the Theory and Applications of Crypto. Techniques*, pages 281–310, 2015.

[48] Pedro Garcia Lopez, Alberto Montresor, Dick Epema, Anwitaman Datta, Teruo Higashino, Adriana Iamnitchi, Marinho Barcellos, Pascal Felber, and Etienne Riviere. Edge-centric computing: Vision and challenges. *SIGCOMM Comput. Commun. Rev.*, 45(5):37–42, September 2015.

[49] Ethereum mainnet statistics. Accessed: 2020-10-05, https://www.ethernodes.org/.

[50] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling Byzantine agreements for cryptocurrencies. In *Proc. 26th Symp. Operating Syst. Principles*, pages 51–68, 2017.
[51] Christian Gorenflo, Stephen Lee, Lukasz Golab, and Srinivasan Keshav. Fastfabric: Scaling hyperledger fabric to 20,000 transactions per second. In *IEEE ICBC*, pages 455–463, 2019.
[52] Rachid Guerraoui, Petr Kuznetsov, Matteo Monti, Matej Pavlovic, and Dragos-Adrian Seredinschi. The consensus number of a cryptocurrency. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, PODC 2019, Toronto, ON, Canada, July 29 - August 2, 2019*, pages 307–316, 2019.
[53] Guy Golan Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael K. Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. SBFT: a scalable and decentralized trust infrastructure. In *Proceedings of the 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2019.
[54] Suyash Gupta, Sajjad Rahnama, Jelle Hellings, and Mohammad Sadoghi. Resilientdb: Global scale resilient blockchain fabric. *Proc. VLDB Endow.*, 13(6):868–883, February 2020.
[55] Timo Hanke, Mahnush Movahedi, and Dominic Williams. Dfinity technology overview series consensus system. Technical report. Accessed: 2021-12-03.
[56] Pavel Khahulin Igor Barinov, Viktor Baranov. POA network white paper, Sept. 2018. https://github.com/poanetwork/wiki/wiki/POA-Network-Whitepaper.
[57] Mike Isaac and Sheera Frenkel. Facebook security breach exposes accounts of 50 million users, Sept. 2018. Accessed: 2020-11-14.
[58] Kristof Jannes, Bert Lagaisse, and Wouter Joosen. You don't need a ledger: Lightweight decentralized consensus between mobile web clients. In *Proceedings of the 3rd Workshop on Scalable and Resilient Infrastructures for Distributed Ledgers (SERIAL)*, pages 3–8, 2019.
[59] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ewa Syta, and Bryan Ford. Omniledger: A secure, scale-out, decentralized ledger via sharding. In *IEEE S&P*, pages 583–598, 2018.
[60] Kadir Korkmaz, Joachim Bruneau-Queyreix, Sonia Ben Mokthar, and Laurent Réveillère. Dandelion: multiplexing byzantine agreements to unlock blockchain performance. Technical Report 2104.15063, arXiv, 2021.
[61] Casey Kuhlman, Benjamin Bollen, Silas Davis, and Dan Middleton. Hyperledger burrow (formerly eris-db), Mar. 2017. Accessed: 2020-11-14, https://www.hyperledger.org/wp-content/uploads/2017/06/HIP_Burrowv2.pdf.
[62] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, July 1982.
[63] Fátima Leal, Adriana E. Chis, and Horacio González-Vélez. Performance evaluation of private ethereum networks. *SN Computer Science*, 1(285), 2020.
[64] Jimmy Lin, Rion Snow, and William Morgan. Smoothing techniques for adaptive online language models: Topic tracking in tweet streams. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, page 422–429, 2011.
[65] Marta Lokhava, Giuliano Losa, David Mazières, Graydon Hoare, Nicolas Barry, Eli Gafni, Jonathan Jove, Rafał Malinowsky, and Jed McCaleb. Fast and secure global payments with stellar. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, pages 80–96, 2019.
[66] Loi Luu, Viswesh Narayanan, Chaodong Zheng, Kunal Baweja, Seth Gilbert, and Prateek Saxena. A secure sharding protocol for open blockchains. In *ACM CCS*, pages 17–30, 2016.
[67] Essam Mansour, Andrei Vlad Sambra, Sandro Hawke, Maged Zereba, Sarven Capadisli, Abdurrahman Ghanem, Ashraf Aboulnaga, and Tim Berners-Lee. A demonstration of the solid platform for social web applications. In *Proceedings of the 25th International Conference on World Wide Web, WWW 2016, Montreal, Canada, April 11-15, 2016, Companion Volume*, pages 223–226, 2016.
[68] Metamask mobile now available on Android and iOS! Accessed: 2020-12-07, https://consensys.net/blog/news/metamask-mobile-now-available-on-android-and-ios/.
[69] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The honey badger of BFT protocols. In *Proc. 2016 ACM SIGSAC Conf. Comput. and Commun. Secur.*, pages 31–42, New York, NY, USA, 2016. ACM.
[70] Mareike Möhlmann and Lior Zalmanson. Hands on the wheel: Navigating algorithmic management and uber drivers' autonomy. In *Proceedings of the International Conference on Information Systems - Transforming Society with Digital Innovation, ICIS*, 2017.
[71] Satoshi Nakamoto. Bitcoin: a peer-to-peer electronic cash system, 2008.
[72] Marshall C. Pease, Robert E. Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *J. ACM*, 27(2):228–234, 1980.
[73] POA Core (POA) Explorer. https://blockscout.com/poa/core.
[74] Jeremias Prassl. *Humans as a Service: The Promise and Perils of Work in the Gig Economy*. Oxford Press, 2018.
[75] Brianna Provenzano. Youtube is down for everyone right now [update: It's back], Nov. 2020. Accessed: 2020-11-14.
[76] Team Rocket. Snowflake to avalanche: A novel metastable consensus protocol family for cryptocurrencies. Technical report, 2018. Accessed: 2021-12-01.
[77] Ilya Sergey, Vaivaswatha Nagaraj, Jacob Johannsen, Amrit Kumar, Anton Trunov, and Ken Chan Guan Hao. Safer smart contract programming with scilla. *Proc. ACM Program. Lang.*, 3(OOPSLA), October 2019.
[78] Gary Shapiro, Christopher Natoli, and Vincent Gramoli. The performance of Byzantine fault tolerant blockchains. In *Proceedings of the 19th IEEE International Symposium on Network Computing and Applications (NCA'20)*. IEEE, Nov 2020.
[79] João Sousa, Alysson Bessani, and Marko Vukolic. A Byzantine fault-tolerant ordering service for the hyperledger fabric blockchain platform. In *48th Annual IEEE/IFIP Int. Conf. on Dependable Syst. and Netw., DSN 2018, June 25-28, 2018*, pages 51–58, 2018.
[80] Chrysoula Stathakopoulou, Tudor David, and Marko Vukolic. Mir-bft: High-throughput BFT for blockchains. Technical Report 1906.05552, arXiv, Sept. 2019.
[81] Andrew S. Tanenbaum and Maarten van Steen. *Distributed systems - principles and paradigms, 2nd Edition*. Pearson Education, 2007.
[82] Pierre Tholoniat and Vincent Gramoli. Formally verifying blockchain Byzantine fault tolerance. In *The 6th Workshop on Formal Reasoning in Distributed Algorithms (FRIDA'19)*, 2019. Available at https://arxiv.org/pdf/1909.07453.pdf.
[83] VMware. Concord. Accessed: 2020-11-28, https://github.com/vmware/concord.
[84] Gauthier Voron and Vincent Gramoli. Technical Report 1912.10367, arXiv, 2021.
[85] G. Wang, S. Wang, V. Bagaria, D. Tse, and P. Viswanath. Prism removes consensus bottleneck for smart contracts. In *2020 Crypto Valley Conference on Blockchain Technology (CVCBT)*, pages 68–77, 2020.
[86] G. Wang, S. Wang, V. Bagaria, D. Tse, and P. Viswanath. Prism removes consensus bottleneck for smart contracts. In *2020 Crypto Valley Conference on Blockchain Technology (CVCBT)*, pages 68–77, 2020.
[87] Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.
[88] Anatoly Yakovenko. Solana: A new architecture for a high performance blockchain v0. 8.13. *Whitepaper*, 2018.
[89] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. HotStuff: BFT consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, New York, NY, USA, 2019. Association for Computing Machinery.
[90] Mahdi Zamani, Mahnush Movahedi, and Mariana Raykova. RapidChain: Scaling blockchain via full sharding. In *ACM CCS*, pages 931–948, 2018.
[91] The zilliqa technical whitepaper. https://docs.zilliqa.com/whitepaper.pdf.

APPENDIX

A. Smart Contract to Reconfigure Nodes

```
pragma solidity ^0.4.0;
pragma experimental ABIEncoderV2;

contract Committee {

     string [] committee;
     address public chairperson;
     uint member;
     mapping(uint => uint) private id;
     mapping(string => bool) hasIp;
     mapping(string => bool) hasCalled;
     mapping(address => string) WallettoIP;
     constructor() public {
       chairperson = msg.sender;
       for (uint d=0; d < 10; d++){
          id[d] = 0;
       }
     }
     uint [] count;
     string [] public selected;
     uint32 [] private digits;
     uint threshold;
     uint size;
     uint select;
     uint option;
     event notify(string []);

     // initial set of node ips and the size of the committee is parse by the chairperson
     function addIp (string [] memory ip, uint members, string[] memory wallets) public {
        committee.length = 0;
         require(
            msg.sender == chairperson,
            "Only chairperson can give right to vote."
        );
        for (uint t = 0; t < ip.length; t++){
           committee.push(ip[t]);
           WallettoIP[parseAddr(wallets[t])] = ip[t];
           hasIp[ip[t]] = true;
           hasCalled[ip[t]] = false;
        }
        size = committee.length;
        member = members;
        // members is the number of participants per committee
     }

     // upon parsing a random seed and once a threshold of calls have been made, a random committee is selected
     function createCommittee (uint val) public {
         // if the caller of this function is in the list of ips added by the chairperson, and if they haven't call this function
         // before - because we don't want t+1 be reached by a malicious node calling this function multiple times
         require(hasIp[WallettoIP[msg.sender]] == true && hasCalled[WallettoIP[msg.sender]] == false);
         hasCalled[WallettoIP[msg.sender]] = true;
         id[val] = id[val] + 1;
         // use size instead of member
         threshold = (size - 1)/3 + 1;
         if (id[val] == threshold){
           select = size/member; // total number of nodes divided by the ones that should be in the committee.
           // gives the possible number of different committees
           for(uint b =0; b<select;b++){
               count.push(b);
           }
           // get the number of options to the range of select
           option = val % select;
           for (uint a = 0; a<count.length; a++){
            if(option==count[a]){
                for (uint j=count[a]*member; j < (count[a] + 1)*member; j++){
                    selected.push(committee[j]);
                }
            }
           }
           emit notify(selected);
           selected.length = 0;
           count.length = 0;
           option = 0;
           for (uint d=0; d < 10; d++){
                id[d] = 0;
```

```
            }
            for (uint num=0; num < size; num++){
                hasCalled[committee[num]] = false;
            }
        }

     }

    // a function to convert addresses of type string to type address
    function parseAddr(string memory _a) public returns (address _parsedAddress) {
        bytes memory tmp = bytes(_a);
        uint160 iaddr = 0;
        uint160 b1;
        uint160 b2;
        for (uint i = 2; i < 2 + 2 * 20; i += 2) {
            iaddr *= 256;
            b1 = uint160(uint8(tmp[i]));
            b2 = uint160(uint8(tmp[i + 1]));
            if ((b1 >= 97) && (b1 <= 102)) {
                b1 -= 87;
            } else if ((b1 >= 65) && (b1 <= 70)) {
                b1 -= 55;
            } else if ((b1 >= 48) && (b1 <= 57)) {
                b1 -= 48;
            }
            if ((b2 >= 97) && (b2 <= 102)) {
                b2 -= 87;
            } else if ((b2 >= 65) && (b2 <= 70)) {
                b2 -= 55;
            } else if ((b2 >= 48) && (b2 <= 57)) {
                b2 -= 48;
            }
            iaddr += (b1 * 16 + b2);
        }
        return address(iaddr);
    }
}
```