

# Dynamic Blockchain Sharding

Deepal Tennakoon 

University of Sydney, Australia

Vincent Gramoli 

University of Sydney, Australia

---

## Abstract

By supporting decentralized applications (DApps), modern blockchains have become the technology of choice for the Web3, a decentralized way for people to interact with each other. As the popularity of DApps is growing, the challenge is now to allocate shard or subnetwork resources to face the associated demand of individual DApps. Unfortunately, most sharding proposals are inherently static as they cannot be adjusted at runtime. Given that blockchains are expected to run for years without interruption, these proposals are insufficient to cope with the upcoming demand.

In this paper, we present dynamic blockchain sharding, a new way to create and close shards on-demand, and adjust their size at runtime without requiring to hard fork (i.e., creating duplicated instances of the same blockchain). The novel idea is to reconfigure sharding through dedicated smart contract invocations: not only does it strengthen the security of the sharding reconfiguration, it also makes it inherently transparent as any other blockchain data. Similarly to classic sharding, our protocol relies on randomness to cope with shard-takeover attacks and on rotating nodes to cope with the bribery of a slowly-adaptive adversary. By contrast, however, our protocol is ideally suited for open networks as it does not require fully synchronous communications. To demonstrate its efficiency, we deploy it in 10 countries over 5 continents and demonstrate that its performance increases quasi-linearly with the number of shards as it reaches close to 14,000 TPS on only 8 shards.

**2012 ACM Subject Classification** Computing methodologies → Distributed algorithms

**Keywords and phrases** Reconfiguration, smart contract, transparency, shard

**Digital Object Identifier** 10.4230/OASICS.FAB.2022.6

**Funding** This research is supported under Australian Research Council Future Fellowship funding scheme (project number 180100496) entitled “The Red Belly Blockchain: A Scalable Blockchain for Internet of Things”.

*Vincent Gramoli*: Australian Research Council

## 1 Introduction

Blockchains, which originally aimed at enabling transparent asset transfers between permissionless individuals [23], has become the de-facto technology for the new version of the World Wide Web, called Web3. In January 2022 alone, the total volume of Web3 sales through decentralized applications (DApps) represented \$16B [19]. These DApps are an appealing alternative to centralized applications, because they offer a **transparent** execution on **secure** data. Unfortunately, DApps create congestions on popular smart contract blockchains, like Ethereum [32]. The key idea to reduce this congestion is called *sharding*, which consists of splitting the workload across disjoint set of computers called shards, subnetworks or zones. In the context of DApps, sharding typically means executing distinct sets of DApps or smart contract functions on different sets of computers [14].

Unfortunately, the existing blockchain sharding protocols (Table 1, later detailed in Section 5) suffer from limitations. In fact, they are typically *static*: once the blockchain is spawned, there is no way to change the number of shards it uses. The problem is that blockchains are intended to run for a long time (e.g., Bitcoin [23] has been running for more than a decade without interruption) whereas new DApps are continuously uploaded to



© Deepal Tennakoon and Vincent Gramoli;

licensed under Creative Commons License CC-BY 4.0

5th International Symposium on Foundations and Applications of Blockchain 2022 (FAB 2022).

Editors: Sara Tucci-Piergiovanni and Natacha Crooks; Article No. 6; pp. 6:1–6:17

OpenAccess Series in Informatics



OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

## 6:2 Dynamic Blockchain Sharding

■ **Table 1** Comparison of sharded blockchains: the dynamism ranges from low, as indicated by ○, to high, as indicated by ●, a checkmark ✓ indicates that the property holds while a cross ✗ indicates that the property does not hold and a dash “–” indicates that it remains unknown.

	Sharded blockchains	Transparency	Dynamism	Shard number dynamism	Shard size dynamism	No synchrony needed
Payments	Elastico [22]	✗	◐	○	◐	✗
	OmniLedger [20]	✗	◐	○	●	✗
	RapidChain [33]	✗	◐	○	●	✗
	SSChain [6]	✗	◐	○	●	✗
DApps	Avalanche [27]	✗	●	●	●	✗
	ChainSpace [1]	✓	◐	○	●	✓
	Cosmos [21]	✗	◐	●	◐	✓
	Eth2 [31]	–	◐	○	◐	✓
	Polkadot [5]	✗	○	○	○	✓
	Zilliqa [29]	✗	○	○	○	✗
	This work	✓	●	●	●	✓

blockchains at runtime. The popularity of these DApps is heterogeneous and a new popular DApp may severely increase the number of requests to a particular smart contract, just like the CryptoKitties DApp that congested the Ethereum network [16]. Ideally, a sharding protocol should allow the blockchain governance to resize the shards and adjust the shard number on-demand without *hard forking*, i.e., creating a duplicated instance of the same blockchain. This would allow to seamlessly migrate DApps from one shard to a newly created one, hence provisioning more resources for popular DApps, grouping less demanded DApps on fewer shards, or offering more resources (e.g., CPU, storage) to a particularly congested shard.

Another problem is that most sharding protocols are *opaque* (cf. 2<sup>nd</sup> column of Table 1): there is no way to securely access their shard configuration. Even if a sharding protocol was made dynamic by offering the users to change the number of shards at runtime, there would be no secure way for these users to confirm the changes took effect. In some cases, sharded blockchains offer a website where users can find information about the current shard configuration. For example, Cosmos [21] offers a website to observe a map of its zones [25]. However, such a web service is typically centralized and prone to a single point of failure, hence defeating the purpose of using a distributed ledger for security. First, this website could simply be hacked, conveying a misleading sharding configuration. Second, the traffic towards the website could be easily redirected with a network attack [12]. Finally, users could expose themselves to phishing attacks by accessing a hacked copy of the website instead of the real one. Such attacks are becoming frequent to fool blockchain users about the information they access online [4].

In this paper, we propose a new dynamic blockchain sharding protocol. As it is intended to operate in open networks, it does not assume synchrony but partial synchrony [11], in that the bound on the message delays is unknown. This protocol is made *transparent* by exploiting the blockchain itself: a minimum number of users can (i) create, (ii) close or (iii) adjust the size of a shard by invoking functions of a smart contract residing on the default shard (called *mainchain*) within a limited time window (if the network asynchrony prevents them from succeeding, then they retry with a larger time window until success).

As all smart contract invocations are logged to the secure storage of the distributed ledger, the shard configuration is securely visible from the world state. Similar to Eth2 [14], a new shard is created as a *shard chain* provisioned by the assets deposited on the mainchain by its users. The most important challenge we had to solve was for the network topology to adapt based on the output of the sharding smart contract: the reconfiguration function emits an event that triggers the spawning, shutdown and restart of some of the blockchain machines.

Like other sharding approaches we provide randomness in shard creation to prevent shard takeovers by malicious nodes. We also present a shard committee rotation approach to mitigate bribery by a slowly-adaptive adversary and a mapping of transactions to shards. Finally, we evaluate our solution on a scalable blockchain called CollaChain [30], which combines DBFT [7], a formally verified [3] consensus protocol, that makes CollaChain fork free; and a Scalable version of the Ethereum Virtual Machine, called SEVM, making it compatible with the largest ecosystem of DApps. Our results confirm that our sharding protocol leads to quasi-linear speedup, that the performance of shards can benefit from a growing number of node resources, and that our mainchain does not act as a performance bottleneck. To summarize, our contribution is threefold:

- We introduce dynamic blockchain sharding, the ability for a blockchain to reconfigure the number of its shards and the size of each of its shards without disrupting the blockchain service. This ability is particularly appealing to cope with the growing of DApps over recent blockchains.
- We propose a dynamic sharding solution that creates a new shard, adjusts a shard, closes a shard, and rotates the shard participants. We implement these algorithms as inherently transparent smart contracts that emit events to replace the current sharding configuration at the network level.
- We demonstrate the feasibility of our approach by implementing our algorithms within a recent scalable blockchain that we deploy in 10 countries across all 5 continents. The experimental results confirm that the performance scales quasi-linearly with the number of shards and demonstrate that the system can achieve close to 14,000 TPS with only 8 shards.

The rest of this paper is ordered as follows: In Section 2 we provide the model and preliminary definitions. In Section 3, we present our dynamic sharding protocol. In Section 4, we illustrate the performance of our solution when deployed at large scale. In Section 5, we discuss the related work. In Section 6, we conclude.

## 2 Preliminaries

### 2.1 Blockchain

A blockchain is a decentralized, distributed system that processes user transactions and logs the transactions to an auditable and cryptographically secure ledger. Each participant keeps a replicated state of the system, and all validator/miner participants require to reach consensus to agree on the set of transactions to be executed. The agreed upon transactions reside in the body of a block data structure and each block has a pointer to the previous block building a chain of blocks known as the blockchain. The header of a block structure consists of a root of a merkel tree known as the state root. The *state root* represents the state of the blockchain at a specific block. Each participant at block  $N$  will have the same state root. By traversing through the Merkle tree from the state root in a block, the accounts, balances, contract data and contract state can be retrieved. A blockchain *committee* is a set

of blockchain participants that execute consensus separately from the rest of the network. In this work, each blockchain committee maintains its own state and process a separate set of transactions.

## 2.2 Model

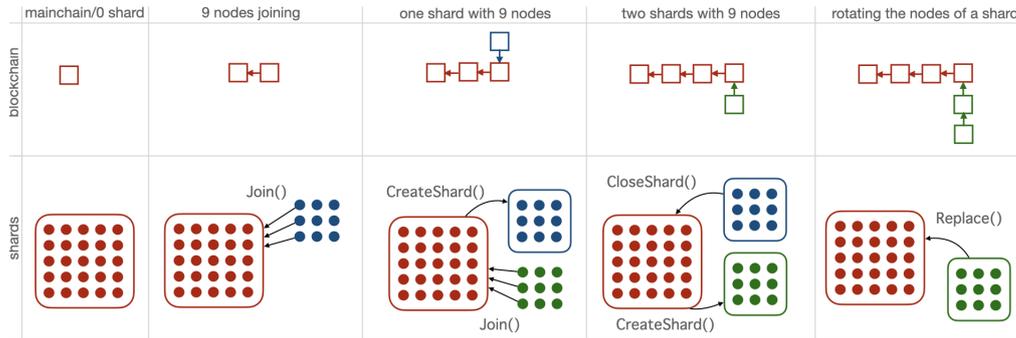
In our system model, we assume our system consists of  $n$  participants. Each participant controls a SEVM and Consensus node in Collachain [30]. Participants join the network in a permission-less manner as outlined in Section 2.3 to tolerate Sybil attacks. From the sample of joined nodes, a group of nodes are selected through a random mechanism (Section 2.3) to the main chain. The set of nodes that execute consensus on the mainchain are termed the *mainchain committee*. Mainchain committee is tasked with administrative tasks of the network such as shard creation, shard committee rotation, and dynamic adjustment of the number of shards and the nodes in a shard. A mainchain can create one or many shards from participants, we term as validator candidates. Each shard keeps separate state, and transactions and is tasked with executing a unique DApp. The shard committee (i.e. the set of validators in the shard) rotates per epoch which signifies a time  $t$  that is sufficiently small to avoid shard takeover by a slowly-adaptive adversary (Section 2.2).

Our network model assumes honest nodes in the network are well-connected and the communication channels between honest nodes are partially synchronous, i.e., if honest nodes broadcast a message, all honest nodes receive it after an unknown time  $T$  and a bounded maximum delay of  $\delta$ . While various sharded blockchains typically assume a stronger property, called synchrony [11], where the upper bound on the delay of every message is known, note that synchrony is typically difficult to guarantee and can easily be violated in an open network like the Internet [12], which has led to numerous double spending attacks against blockchains [24, 13].

In our threat model, out of  $n$  participants, we assume  $f$  are byzantine such that  $f < n/4$ . This is to ensure that a committee has  $k$  nodes such that  $f_k < k/3$  with high probability, similar to previous work [20, 22]. Note that  $n$  and  $k$  can vary at run-time due to the dynamism to our approach. Only 1/3 of the mainchain nodes can be byzantine. The byzantine nodes can behave arbitrarily or collude to attack the system. All correct nodes adhere to the presented protocol (Section 3). In order to cope with a *Sybil attack*, whereby an adversary forges fake identities to outnumber significantly the participants identities, we use a proof-of-stake (PoS) mechanism as described in the subsequent section. A bribery consists of an adversary incentivizing a committee participant to join its coalition. A *shard-takeover attack* consists of an adversary gaining control over sufficiently many nodes within a committee to prevent consensus from being reached. As explained below we assume a slowly-adaptive adversary that can bribe all nodes but only progressively (not instantaneously) [20, 33] and we cope with this attack by proposing a rotating committee.

## 2.3 Bootstrapping

We consider a *permissionless* model. Any participant can join or leave the network without permission. Our membership protocol is similar to Algorand [18] where participants that require to be a part of consensus needs to stake some coins. A weight is assigned to each joining participant based on their stake. Consequently, a subset of participants are selected to perform consensus on the mainchain based on a random beacon and the weights of the nodes. This helps prevent Sybil attacks. To mitigate bribery take-overs, the main chain committee rotates periodically and the size of the main chain is changeable in a similar



■ **Figure 1** An example of the consecutive steps (from left to right) of a dynamic sharding execution where 2 shards are created, one of these shards is closed and shard nodes are rotated.

approach to how we change the shard size (Algorithm 1 line 15), which allows new nodes to join the main chain committee if a threshold of mainchain participants agree. We do not implement our membership protocol in this paper but leave it as a part of future work.

### 3 The Dynamic Sharding Protocol

In this section, we present how the dynamic sharding protocol adjusts the size and number of shards, and how it rotates shard nodes.

#### 3.1 Overview

Figure 1 depicts a high level example of a dynamic blockchain sharding execution where smart contract invocations stored in blocks reconfigure the sharding. Initially, there are 25 participants in the mainchain with a single genesis block, as depicted on the 1<sup>st</sup> column, they decide the shard size. Then, external participants invoke the `Join(·)` function to join a new shard (cf. 2<sup>nd</sup> column). When enough of them have joined, the `CreateShard(·)` function is invoked on the mainchain smart contract and creates a new (blue) shard (cf. 3<sup>th</sup> column). The resulting function invocation is stored as a transaction of a new block of the mainchain. A new (blue) shard chain, maintained by the shard is created: it is linked to the block of the mainchain where its creation invocation is stored. New participants invokes the `Join(·)` function as depicted in the middle column. After that, the `CreateShard(·)` function creates a new (green) shard while the old (blue) shard invokes the `CloseShard(·)` function, which reports the blue shard history to a new block of the mainchain (cf. 4<sup>th</sup> column). Finally, the new shard rotates its participants by executing the `Replace(·)` function whose invocation gets stored in a new block.

#### 3.2 Shard creation

The shard creation is presented in Algorithm 1 and is deployed on the mainchain as a smart contract during the bootstrap of the blockchain. The variable *admins* keeps track of a set of mainchain participants and *NumberOfAdmins* refers to the number of participants in the mainchain.

The shard creation smart contract is initialized with a set of data structures. The variable *event* refers to a broadcast message sent to all blockchain nodes in a shard. The event has a name (e.g., `ShardNodes`) and values that it broadcast (i.e. `ShardNodes` broadcasts an unsigned

■ **Algorithm 1** The smart contract that triggers the creation of a new shard.

---

```

1: Initialization:
2:  event ShardNodes(uint, string[], address[])
3:  uint shardSize
4:  uint NumAccounts
5:  mapping (string → string[]) shard
6:  mapping (address → bool) called
7:  mapping (string → bool) voted
8:  mapping (uint → address[]) accounts
9:  mapping (uint → uint) SizeOfShard
10: mapping (uint → uint) NumberOfShards
11: mapping (uint → bool) Created
12:  admins : the set of addresses of admins
13:  NumberOfAdmins = | admins |
14:
15: SetSize(val, NShards):                                ▷ threshold of admins set shard size, number & accounts/shard
16:   if SenderAddr ∈ admins then                          ▷ if function invoker is an admin
17:     SizeOfShard[val]++
18:     NumberOfShards[NShards]++
19:     if NumberOfShards[Shards] == (2 * NumberOfAdmins / 3 - 1) &
20:        SizeOfShard[val] == (2 * NumberOfAdmins / 3 - 1) then  ▷ if shard size, number, accounts agreed
21:       shardSize = val
22:       NumShards = NShards
23: JoinShard(ipAddr):                                    ▷ when a node wants to join a shard as validator
24:   if called[senderAddr] == false & voted[ipAddr] == false then  ▷ avoid assigning IP twice to shard
25:     called[senderAddr] = true
26:     voted[ipAddr] = true
27:     random = RANDContractAddr.GetRand()                 ▷ Fetch random number from RANDAO
28:     shard[random mod (NumShards)] ∪ ipAddr
29:     accounts[random mod NumShards] ∪ senderAddr
30:     if length(shard[random mod (NumShards)]) == shardSize & Created[tag] == false then
31:       CreateShard(random mod (NumShards), shard[random mod (NumShards)],
32:         accounts[random mod (NumShards)])
33:       length(shard[random mod (NumShards)]) = 0        ▷ reset the shard tag value to 0
34:       length(accounts[random mod (NumShards)]) = 0     ▷ reset the shard tag value to 0
35:
36: ClosedShards(tag):                                    ▷ admin calls this if received n-t COMMITS for close from shard chain
37:   Closed[tag] = true
38:
39: CreateShard(tag, [] shard, [] accounts):
40:   emit ShardNodes(tag, shard, accounts)                ▷ emit ip addresses & accounts of shard nodes, triggers shard start
41:   Created[tag] = true                                   ▷ Assign shard as created

```

---

integer, a string array and an address array). A *mapping* is a data structure mapping a key to a value. The *bool* is a boolean data type and  $| \text{admins} |$  is the set containing the wallet addresses of admin nodes.

Admins of the main chain start by setting the size of shards and the number of shards (Algorithm 1 line 15). Note that a threshold of admins should agree to the same settings for these values to be set (Algorithm 1 line 19), and a threshold of admins can again agree to change these values during run time making the sharding dynamic. Note that unlike any other sharding scheme, we provide the capability to change shard size and number of shards even when the number of nodes in the network remains constant (no nodes joining or leaving).

The validator candidates invoke the `JoinShard` function and parse their IP address (Algorithm 1, line 23) in an attempt to join a shard. Note that, at line 24 of Algorithm 1, prevents validator candidates from joining multiple shards as well as two validator candidates from joining shards with the same IP address. Consequently, the shard creation

contract fetches a random number *random* from a verifiable random number generation contract (Algorithm 1, line 27) taken out of our system. (We rely here on the RANDAO implementation [26] of a random number generator that is expected to be used in Ethereum 2.0 as an example only. Randao is synchronous but a partially-synchronous random number generation solution can easily be used instead [9]).

Based on *random*, the IP address of a validator candidate is assigned to a random key of a *shard* mapping (Algorithm 1, line 28). Deriving the key values as:  $random \bmod NumShards$  ensures that the IP address of a candidate is assigned a shard tag  $x$  such that  $x \in \{0, 1, \dots, NumShards\}$ .

Similarly, wallet address of the validator candidate is also added to a random key corresponding to a shard tag of an *account* mapping.

We underscore that *NumShards* can be adjusted by admins to accommodate more, or less shards in the system. If for a particular shard key/tag the maximum number of nodes (i.e., *ShardSize*) that could be assigned is complete, a *CreateShard* function is invoked, parsing an array of validator candidates and validator accounts for a shard tag (Algorithm 1, line 32).

The *CreateShard* function, emits a smart contract event *ShardNodes* (i.e., a broadcast to all participants) with the validator IP addresses and accounts that should be in a particular shard tag (Algorithm 1, line 32).

Validator candidates upon receiving *ShardNodes* event verifies its IP address is included in the event. If included, the validator candidates reconfigure and form a validator committee for a shard with a specific tag. Details of this process is outlined in Algorithm 2.

■ **Algorithm 2** The algorithm executed by a participant to create a new shard upon reception of the smart contract creation event.

---

```

1: Upon receiving a smart contract event:
2:  event ← subscribe(CreateContractAddr)  ▷ all nodes subscribe to events from shard create smart contract
3:  if localIP ∈ event then  ▷ If local IP is in event
4:    tag, shard, accounts ← extract(event)  ▷ extract values from event
5:    stop(node)
6:    editGenesis(accounts)  ▷ Edit genesis with accounts
7:    connectPeers(shard)  ▷ connect with other members of the shard
8:    start(node)

```

---

### 3.3 Shard closing

Shard closing is a procedure that helps prevent resource wastage. If a shard is not processing many transactions or is idle for a while a shard nodes can decide to close the shard. This is a part of the extended dynamism our protocol provides.

Algorithm 3 presents the smart contract algorithm for closing a blockchain shard in a partially synchronous manner. The variable  $N_v$  is the number of nodes that the shard contains.

Firstly, once a shard node decides to close the shard it is a part of, it invokes the *CloseShard* (Algorithm 3 line 10) parsing the state root the node prefers to close at. Algorithm 3 line 11-line 12, ignores if a state root is parsed to the function by participants more than once. Otherwise, the *threshold* is increased (Algorithm 3, line 14), which indicates the number of participants that have parsed a specific state root to the *CloseShard* function. At line 15 of Algorithm 3, if  $2 \cdot N/3 + 1$  nodes s.t.  $N$  is the total number of participants in the shard have parsed the same state root to *CloseShard*, then a COMMIT event is emitted with the *ShardTag*. Otherwise, the parsed state root to the *CloseShard* function is emitted in a *Bs* event.

■ **Algorithm 3** The smart contract that triggers the closing of a new shard.

---

```

1: Initialization::
2:   event Bs(string y)
3:   event COMMIT(string x, string m)
4:   mapping(string → uint) threshold
5:   uint Nv
6:   bool reached
7:   mapping(bytes32 → string) called;
8:   Nv = val
9:   ShardTag = tag
10: CloseShard(stateroot):
11:   if called[hash(SenderAddr, stateroot)] == true then
12:     return
13:   called[hash(SenderAddr, stateroot)] = true
14:   threshold[stateroot] = threshold[stateroot] + 1
15:   if threshold[stateroot] == 2*Nv/3+1 & !reached then
16:     reached = true
17:     emit COMMIT(stateroot, "COMMIT", ShardTag)
18:   emit Bs(stateroot)

```

▷ *The number of nodes in a shard from Algorithm 1*  
 ▷ *tag of shard generated– based on RANDAO in Algorithm 1*  
 ▷ *nodes call CloseShard parsing the state root*  
 ▷ *SenderAddr parsed stateroot before*  
 ▷ *avoids double voting*  
 ▷ *'SenderAddr' parsed stateroot*  
 ▷ *number of nodes parsed specific 'stateroot'*  
 ▷ *state root first reaching threshold*  
 ▷ *emits a commit event with the stateroot*  
 ▷ *emits event with the parsed stateroot*

---

■ **Algorithm 4** The algorithm executed by a participant to close a shard upon reception of the smart contract closure event.

---

```

1: Upon receiving a smart contract event:
2:   event ← subscribe(CloseContractAddr)
3:   if contains(event, COMMIT) then
4:     number = getCurrentBlockNumber()
5:     for i = 0; i < number; i++ do
6:       block ← getBlock(i)
7:       if block.stateroot = event.stateroot then
8:         Close(block.number)
9:       else
10:        if nodeHas(event.stateroot) then
11:          closeContractAddr.CloseShard(event.stateroot)
12:        else
13:          pending.push(event.stateroot)
14:          Check()
15:
16: Check():
17:   for i=0; i < length(pending); i++ do
18:     if nodehas(pending[i]) then
19:       CloseContractAddr.CloseShard(pending[i])

```

▷ *all nodes subscribe to closing smart contract*  
 ▷ *smart contract event contain the "COMMIT" string*  
 ▷ *parse closing block number to sync balances algo*  
 ▷ *exit code*  
 ▷ *If node has same state root*  
 ▷ *pass stateroot to SC*  
 ▷ *push the stateroot to a pending array*  
 ▷ *Do in parallel*  
 ▷ *parse stateroot to smart contract*

---

Algorithm 4 presents the execution at a shard participant when either a COMMIT or a *Bs* smart contract event is received from the shard close smart contract algorithm (Algorithm 3).

A shard participant subscribes to the close shard smart contract in its state. Upon receiving a smart contract event from this smart contract (*CloseContractAddr*) at the shard node, the *event* is filtered. Consequently, the shard participant checks if the event is a COMMIT event (i.e., whether it contains the keyword COMMIT) at Algorithm 4 line 3. If this condition is met, the current block number (Algorithm 4 line 4) of the participant is retrieved and the state of the shard participant is traversed from the 0<sup>th</sup> block to the current block to find the block number that contains the state root. If a block exists with the received state root in the shard node, it decides to parse the block number to a Close function (Algorithm 4 line 8) shown in Algorithm 5 and exits.

If the *event* is not of type COMMIT but the shard participant has the state root contained in the event (Algorithm 4 line 11), the participant invokes the CloseShard function in the Close shard smart contract parsing the state root. If the *event* is not of types COMMIT and

the shard node does not have the state root received, it is pushed to a pending array and kept (Algorithm 4 line 14), in case the shard participant sees the state root sometime in the future. In Algorithm 4 line 16, a *Check* function concurrently and repeatedly checks, if the shard node has the pending state root. The *CloseShard* function is invoked parsing the state root if the state root is found (Algorithm 4 line 19).

■ **Algorithm 5** Shard chain participant Send Closing Account Balances to main chain.

---

```

1: Initialization:
2:    $A$  is the set of account addresses in the shard
3:    $Account(address, balance)$  ▷  $A$  tuple of address and balance
4:    $SA$  is the set of  $Account(address, balance)$  tuples

5:  $Close(BNumber)$ : ▷ parse block number at which the shard should close
6:   for  $a \in A$  do
7:      $b \leftarrow getBalance(a, BNumber)$  ▷ Balance of account  $a$  at closing block
8:      $SA \cup Account(a, b)$ 
9:    $Broadcast(SA, ShardTag)$  ▷ Broadcast to main chain nodes
10:   $stop(shardNode)$ 

```

---

Algorithm 5 executes at each shard participant and retrieves balances of all accounts at the block number that the shard closes (Algorithm 4 line 8) and broadcasts it and the shard tag to the main chain participants (Algorithm 5 line 9). Note that this broadcast is a reliable broadcast and waits for an ACK before the shard participants stop in the subsequent line.

■ **Algorithm 6** Syncing of balance at the main chain from shard chains .

---

```

1: Initialization:
2:    $threshold = 2N/3 + 1$  s.t.  $N$  is the total number of shard chain nodes
3:   mapping (bytes32  $\rightarrow$  uint)  $count$ 

4:  $Receive(SA), ShardTag$ : ▷ Receive Account tuple set
5:    $count[hash(SA)] \leftarrow count[hash(SA)] + 1$  ▷ times specific account tuple set received
6:   if  $count[hash(SA)] == threshold$  then ▷ If threshold of same  $SA$  received
7:      $CreateContractAddr.ClosedShard(ShardTag)$ 
8:      $stop(node)$ 
9:      $editGenesis(SA)$  ▷ Edit the genesis, adding accounts and balances tuple set
10:     $start(node)$ 

```

---

A main chain participant upon receiving the tuple set of accounts and balances  $SA$  from shard participants, and the shard tag, executes Algorithm 6. Upon receiving  $SA$ , the algorithm keeps count of the number of unique  $SA$  sets received (Algorithm 6 line 5). If  $2 \cdot N/3 + 1$  number of the same  $SA$  set is received s.t.  $N$  is the number of participants in the closing shard, the mainchain node invokes the *ClosedShard* function in Algorithm 1 to set the shard with the specific *tag* as closed. Consequently, the main chain participants stop (Algorithm 6 line 8), edits the genesis adding the new accounts and balances (Algorithm 6 line 9) and restarts (Algorithm 6 line 10). This way, the mainchain participants are synced with the accounts and balances of the shard chain. Note that, syncing accounts and balances from multiple shard chains upon shard closing does not affect the consistency of the state in the main chain since accounts in each shard are disjoint.

### 3.4 Shard committee rotation

A shard with a particular tag remains active once it is created until it is closed. There is a risk of participants being bribed by a slowly-adaptive adversary while a shard is active. If sufficiently many participants in a shard committee are bribed this way (at least  $1/3$ ), there is

## 6:10 Dynamic Blockchain Sharding

a risk of shard takeover. To mitigate this risk, we propose a shard committee rotation protocol that is part of the shard creation smart contract (Algorithm 1) but presented separately below for clarity. We consider an epoch as a specific time  $t$  where a shard committee processes transactions. At every  $t$  interval, all correct shard participants performs committee rotation correctly. Note that the number of correct nodes in a shard is greater than  $2N_v/3$ .

■ **Algorithm 1 Extension** Shard committee rotation algorithm, a part of shard creation smart contract.

---

```
42: Initialization::
43:   mapping (string → uint) ReplaceIpInvoked
44:   mapping (address → uint) ReplaceAddressInvoked

45: Replace(ipAddr, tag): ▷ Shard node parses its Ip address
46:   ReplaceIpInvoked[ipAddr]++
47:   ReplaceAddressInvoked[senderAddr]++
48:   if ReplaceIpInvoked[ipAddr] >  $2 \cdot N_v/3$  & ReplaceAddressInvoked[senderAddr] >  $2 \cdot N_v/3$  then
49:     called[senderAddr] = false
50:     voted[ipAddr] = false
51:     Created[tag] = false
52:     JoinShard(ipAddr) ▷ Invoke JoinShard in Algorithm 1
```

---

The committee rotation starts with shard participants invoking **Replace** in the Algorithm 1 Extension at line 45. Each correct shard participant parses the IP address of each of its committee members and their shard tag simultaneously to the **Replace** function. If a particular IP address and sender address has been used for the invocation  $2 \cdot N_v/3$  times, the *called* and *voted* mappings are set to false for the corresponding IP address and sender address. Consequently, the *Create*[*tag*] is set to false and the **JoinShard** function is invoked parsing the IP address. The **JoinShard** function ensures shard participants are assigned to new shard committees following the same process of shard creation, that is, rotating the shards committees every epoch. Note at the end of an epoch before the committee is rotated, the mainchain participants can adjust the number of shards and the number of members per shard parameters according to the workload, which will change the number of shards and the nodes per shard, making our sharding approach dynamic.

### 3.5 Transaction assignment

In a web-scale blockchain that we foresee, each DApp executes on at most one shard. This concept is known as application or service-oriented sharding [17]. For example, we would have a Twitter DApp on one shard, a Youtube DApp on another shard. Each client sends requests to the shard that executes their required DApp. Each shard tag and the services they execute will be made available publicly so the clients can connect to the shard they prefer to send their transactions. Due to the service-oriented nature of sharding the state of each shard is disjoint, hence state consistency is not affected due to data migrations happening from shard closing and shard rotation of multiple shards. Also, due to the shard independence there is no need for cross-shard transactions. We do not present our own cross-sharding protocol and it is out of the scope of this paper.

### 3.6 Availability

Committee rotation in every epoch is essential to tolerate a slowly-adaptive adversary. However, frequent changes of committees is a challenge when the state is sharded. A new shard committee, needs to sync the state from a previous shard committee, to service the

DApps for a particular shard tag. This syncing process involves downloading the entire blockchain from previous nodes and is an expensive task, which is known to bottleneck performance and affect the availability of shard nodes for transaction processing [6]. Since our sharding approach is byzantine fault tolerant downloading the latest state would suffice by querying  $f + 1$  previous shard committee members. There is no need to download the entire state history (i.e. snapshots) nor the entire block history. As such, we provide better availability than some sharding approaches that shard the state as well [20].

### 3.7 Proof sketches

The first lemma shows that each shard contains less than  $N_v/3$  byzantine participants with high probability. This is key to guarantee agreement among each shard to guarantee that the view of the blockchain is consistent across all replicas. For simplicity in the analysis, we assume that the shard participants correspond to a sample of  $N_v$  participants taken uniformly at random among the whole set of  $n$  participants and we reuse the same reasoning as in [22].

► **Lemma 1.** *In each shard of  $N_v$  participants, there are less than  $N_v/3$  byzantine participants with high probability.*

**Proof.** By assumption we return a participant taken uniformly at random among all  $n$  participants. Consider each of this event as a Bernoulli trial such that a random variable  $X_i$  is 1 if the returned participant is correct and 0 if it is byzantine. Let  $\rho$  be the portion of byzantine participants. Because there are at most  $f < n/4$  byzantine participants among the initial  $n$  participants, we have  $\rho < 1/4$ .

$$\begin{aligned}\Pr[X_i = 1] &= p = \frac{(n-\rho)}{n}, \\ \Pr[X_i = 0] &= 1 - p = \frac{\rho}{n}.\end{aligned}$$

The random variable  $X = \sum_{i=1}^{N_v} X_i$  thus follows a binomial distribution and  $\Pr[X = k] = \binom{N_v}{k} \rho^{N_v-k} (1-\rho)^k$ , hence we can derive the probability  $\Pr[X \leq 2N_v/3]$  of creating a shard with less than  $2/3$  of correct participants:

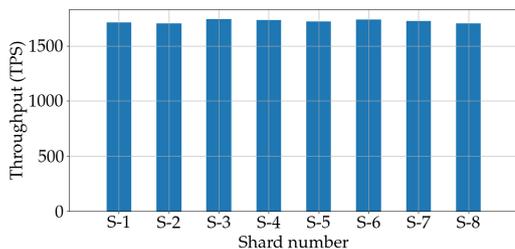
$$\Pr[X \leq 2N_v/3] = \sum_{k=0}^{2N_v/3} \binom{N_v}{k} \rho^{N_v-k} (1-\rho)^k.$$

As this probability decreases exponentially fast with  $N_v$  there exists a parameter  $\lambda$  and a constant  $n_0$  for which  $\Pr[X \leq 2N_v/3] \leq 2^{-\lambda}$  for all  $N_v \geq n_0$ . As a result, each shard contains at most  $\lceil N_v/3 \rceil - 1$  byzantine participants with high probability, which concludes the proof. ◀

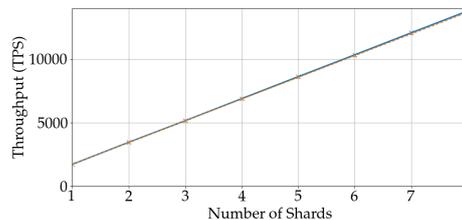
Given Lemma 1 and that our protocol relies on the DBFT [7] consensus protocol, which is resilient optimal, we know that participants agree when less than  $N_v/3$  are byzantine. Hence each time a new block is added to a shard that did not fail, then the shard remains consistent with high probability. As a result, the transparent access to sharding information remains guaranteed. An important remark is that the proof of Lemma 1 relies on having  $N_v \geq n_0$ , however, for the sake of the empirical analysis we choose  $N_v$  relatively small (up to 60 machines) in Section 4 to limit the cost of our AWS experiments.

► **Lemma 2.** *If  $2 \cdot N_v/3 + 1$  of the participants of shard  $s$  invoke its `CloseShard()` function with the same argument, then the shard  $s$  eventually closes.*

## 6:12 Dynamic Blockchain Sharding



■ **Figure 2** Throughput per shard.



■ **Figure 3** Linear increase in throughput with increasing number of shards.

**Proof.** The state root at which a shard participant wishes to close the shard is received by all correct participants in the shard by the  $Bs$  event (Algorithm 3, line 18) since the network is partially synchronous. Also, if a participant agrees to close the shard at a particular state root after seeing the state root event, they will either have that state root in their history or will eventually have it since consensus ensures the nodes have the same state history eventually. Therefore, if at some point in time,  $2 \cdot N_v/3 + 1$  participants (Algorithm 3 line 15) – where the number of byzantine participants is  $f < N_v/3$ , agree on the state root, a commit event will be emitted (line 17) triggering the close of the shard. ◀

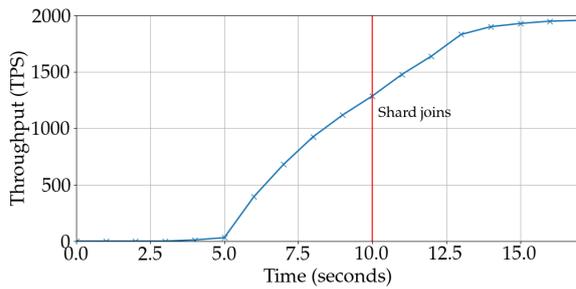
## 4 Evaluation

In this section, we evaluate the performance and the dynamism of our sharding solution. Our sharding approach was implemented on Collachain [30], a DApp supported blockchain. Note that while we evaluated on Collachain to benefit from the fork-free guarantees, our solution is adaptable for any Ethereum-based blockchain should the fork-free guarantees not be needed. We implemented our smart contract algorithms using Solidity and algorithms running at participant nodes using Web3js. All the experiments were performed on AWS, with c5.4xlarge (16 vCPUs, 32GB RAM) blockchain instances (i.e., which have similar performance to a modern PC) and c5.xlarge (4 vCPUs, 8GB RAM) client instances sending asset transfer transactions. A balanced workload was sent to each shard.

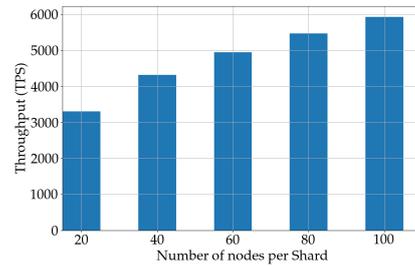
### 4.1 Dynamic shard adjustment

Figure 3 presents the scalability of our solution. Each shard consists of 60 machines evenly distributed across 10 AWS regions spanning 5 continents: Ohio, Mumbai, Seoul, Singapore, Sydney, Tokyo, Canada, Frankfurt, London, Paris, Stockholm, São Paulo. The dotted line is a straight line indicating the ideal speedup one could expect from multiplying the performance of the first shard by the number of shards. The continuous line represents the throughput with the number of shards. As we can see, the throughput increases almost linearly with the number of shards. At 8 shards, the throughput is 13,808 TPS. The quasi-linear growth in throughput is expected since, each shard processes a unique set of transactions, without performing cross-shard transactions because, as mentioned previously, our shards are dedicated to independent DApps. Therefore, the throughput of the entire network at 8 shards almost equals the sum of the throughputs of all shards.

Due to the dynamism of our sharding approach, the number of shards can be varied by creating or closing shards at runtime. Figure 4 depicts the throughput over time when new shards are created. Each shard consists of 8 machines and was evaluated in the Sydney AWS



■ **Figure 4** Throughput over time when new shards join.



■ **Figure 5** Throughput of 3 shards as their size increases.

region. At 0 second, there is only 1 shard. At around 10 seconds, another shard is created and starts processing transactions. Before the curve flattens and because a new shard starts processing transactions, the throughput keeps growing. Finally, the throughput stabilizes as expected when both shards keep processing transactions at full capacity. Note that the throughput of the mainchain is not considered as it only performs administrative tasks such as shard creation and shard rotation and does not process client transactions.

## 4.2 Dynamic node adjustment

With the dynamism we provide, the number of nodes in a shard can also be adjusted at runtime after an epoch time period, even when the total number of nodes in the network remains constant. Figure 5 illustrates this capability: We keep the number of shards fixed to 3 and vary the amount of nodes per shard. As can be seen, when the number of nodes per shard increases from 20 to 100, the total throughput also increases. At 50 participants per shard, a throughput of  $\sim 6000$  TPS is achieved. As CollaChain is known to be scalable [30] in that its performance grows with the provided resources, so does the throughput with the increasing number of nodes here. This makes our sharding approach particularly suited to run on top of CollaChain, so as to achieve dynamism while maintaining performance. Note that, the number of nodes per shard could have been increased further while maintaining performance due to CollaChain scalability.

## 5 Related Work

In this section, we present works related to blockchain sharding and previously summarized in Table 1. Section 5.1 lists the sharding protocols of blockchains offering native transfers of assets while Section 5.2 lists the sharding protocols of blockchains supporting smart contract, and thus DApp, execution. Interestingly, even the protocols for blockchains that support smart contracts do not invoke smart contract functions to reconfigure their shards. Some interesting works already create shards based on attributes (like locations [2]) while others rotate shards with randomization [8] like we do, however, we focus below on dynamism.

### 5.1 Payment Blockchains

Elastico [22] is the first sharded permissionless blockchain that tolerates byzantine failures. Elastico assumes synchrony and that at most  $1/4$  of the computational power is owned by byzantine participants. It mitigates Sybil attacks and shard take-overs with proof-of-work (PoW) and randomness, respectively, and rotates committees to tolerate static and

round-adaptive adversaries launching bribery attacks. Elastico upper bounds the number of validators per committee to 100, indicating a partial shard size dynamism, but it requires the number  $k$  of committees to be adjusted offline, which limits shard number dynamism. We are not aware of any mechanism to audit Elastico’s current sharding configuration, like the number of validators.

OmniLedger [20] improves upon Elastico’s decentralization and high failure probability and offers higher performance. Like for Elastico, Omniledger assumes synchrony, offers Sybil resistance via randomness and does not allow to audit the sharding configuration or to change the shard number at runtime. Omniledger rotates validators in each epoch using cryptographic sortition and a verifiable random function to mitigate bribery attacks. In addition of shards running their own instance of consensus, Omniledger also shards the blockchain state. Unlike Elastico, OmniLedger does not limit the number of validators, hence offering a higher degree of dynamism, yet it does not communicate transparently the number of validators to its users.

RapidChain [33] is the first sharded blockchain to support up to  $f < n/3$  byzantine failures where  $n$  is the number of participants. Like Omniledger, RapidChain assumes synchrony and lets each shard maintain a portion of the blockchain state and run its own consensus instance. Candidate nodes solve a proof-of-work puzzle and create identities that they send to a reference committee, which randomly defines the next epoch committees. RapidChain allows nodes to join and leave the network and assigns them to existing shards, hence it offers a static shard number but a dynamic shard size.

SSChain [6] avoids the rotation of shard committees to shard the state without having to download the blocks and state. These data migrations, needed to verify transactions, can severely impact availability of the sharded blockchain. SSChain changes shards by allowing nodes to freely join, however, the risk is for a byzantine coalition to take over a shard. SSChain mitigates this attack by introducing a two-chained approach: a root chain verifies the blocks coming from each shard before committing them, which provides safety despite shard take-over at the condition of maintaining the entire state. SSChain offers neither transparency nor shard number dynamism but offers shard size dynamism.

## 5.2 DApp supported blockchains

Ethereum 2.0 (Eth2) is expected to introduce sharding to improve Ethereum’s performance. Eth2 contains a fixed set of 64 shard chain and a single beacon chain [15]. Our approach is similar to Eth2 since we also request validators to escrow a deposit on the mainchain before assigning them to shard chains. However, Eth2 requires a minimum of 111 validators [31] to lower the probability of  $2/3$  adversarial nodes in a shard to  $2^{-40}$ . At the end of each epoch, validators are rotated to maintain availability despite a slowly-adaptive adversary. Each shard runs a series of 64 Casper FFG consensus instances per epoch, after which a new block containing the shard states is appended to the beacon chain. Our approach differs from Eth2 by not forking, thanks to DBFT [7], not assuming synchrony, executing smart contracts even in the mainchain, and offering transparency and dynamism.

ChainSpace [1] is a transparent sharded blockchain that does not assume synchrony. An admin contract maps other smart contracts or “objects” to nodes that function as a shard, hence allowing users to consult the sharding configuration without inconsistencies. ChainSpace requires the admin contract creator to be trustworthy because if a shard contains a too large byzantine coalition, then the state of the blockchain could be compromised. ChainSpace offers transparency of the sharding configuration to its users and allows to dynamically adjust the number of nodes per shard, but cannot change the number of shards at runtime.

Zilliqa [29] exploits PoW and a random beacon to maintain two committees: one committee, called the “DS committee”, is elected with PoW to create shards. Two pseudo-random numbers are generated:  $r1$  comes from the last block in the previous DS committee while  $r2$  comes from the last transaction block in a shard. The nodes solve an Ethash PoW cryptopuzzle based on their private key  $P_k$ , IP,  $r1$  and  $r2$ . The first to solve this puzzle proposes a block that the DS committee agrees upon. Consequently, the successful miner is added to the DS committee and the oldest miner is churned out. At all times the DS committee has the most recent  $n$  miners. Zilliqa does not shard the state, assumes network synchrony and does not provide sharding dynamism or transparency.

Avalanche [27] offers “subnets” that can be viewed as shards. Three default subnets run three separate blockchains, the P-Chain handles metadata, the C-Chain handles native payments and the X-Chain handles smart contract executions. Avalanche offers dynamism because new subnet can be created and new validators can be added to an existing subnet. Unfortunately, Avalanche cannot work in a partially synchronous setting because its participants have to wait for the response of a small sample of nodes to progress, which could all be faulty [28]. Avalanche offers a JSON API to retrieve information about subnets, however, we are not aware of any verifiable way to collect tamper-proof information.

Cosmos [21] is a network of “zones” that can be viewed as shards as well. Each zone is a separate blockchain and the main one, called Hub, manages the governance of this network. Each zone builds upon the Tendermint consensus protocol that assumes partial synchrony and requires  $f < n/3$  to solve consensus. Zones are not fully dynamic in that there cannot be more than a maximum of validators per zone, seemingly because performance decreases with the Tendermint participants. Even though the maximum number of validators per zone is announced to grow from 100 to 300 over a period of 10 years, one cannot add validators beyond this point. Although Cosmos offers information about validators [10] and zones [25] we are not aware of any way to guarantee this information is correct, as this information is not stored in the cryptographically secure ledger.

## 6 Conclusion

In this paper, we introduced dynamic blockchain sharding, the ability for a blockchain to change its sharding configuration at runtime without hard forks. Our implementation relies on smart contracts, hence anyone can double check the effectiveness of the reconfiguration by auditing the current state of the blockchain. The performance of our world-wide geodistributed setting demonstrates that dynamic sharding scales quasi-linearly and can offer close to 14,000 TPS with only 8 shards.

---

### References

- 1 Mustafa Al-Bassam, Alberto Sonnino, Shehar Bano, Dave Hrycyszyn, and George Danezis. Chainspace: A sharded smart contracts platform. *arXiv preprint*, 2017. [arXiv:1708.03778](https://arxiv.org/abs/1708.03778).
- 2 Mohammad Javad Amiri, Divyakant Agrawal, and Amr El Abbadi. *SharPer: Sharding Permissioned Blockchains Over Network Clusters*, pages 76–88. Association for Computing Machinery, New York, NY, USA, 2021. [doi:10.1145/3448016.3452807](https://doi.org/10.1145/3448016.3452807).
- 3 Nathalie Bertrand, Vincent Gramoli, Igor Konnov, Marijana Lazić, Pierre Tholoniati, and Josef Widder. Brief announcement: Holistic verification of blockchain consensus. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, July 2022.
- 4 Russell Brandom. \$1.7 million in nfts stolen in apparent phishing attack on opensea users. Accessed: 2022-03-11. URL: <https://www.theverge.com/2022/2/20/22943228/opensea-phishing-hack-smart-contract-bug-stolen-nft>.

- 5 Jeff Burdges, Alfonso Cevallos, Peter Czaban, Rob Habermeier, Syed Hosseini, Fabio Lama, Handan Kilinc Alper, Ximin Luo, Fatemeh Shirazi, Alistair Stewart, and Gavin Wood. Overview of polkadot and its design considerations, 2020. [arXiv:2005.13456](https://arxiv.org/abs/2005.13456).
- 6 Huan Chen and Yijie Wang. Sschain: A full sharding protocol for public blockchain without data migration overhead. *Pervasive and Mobile Computing*, 59:101055, 2019. doi:10.1016/j.pmcj.2019.101055.
- 7 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.
- 8 Hung Dang, Tien Tuan Anh Dinh, Dumitrel Loghin, Ee-Chien Chang, Qian Lin, and Beng Chin Ooi. Towards scaling blockchain systems via sharding. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD '19*, pages 123–140, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3299869.3319889.
- 9 Luciano Freitas de Souza, Sara Tucci-Piergiovanni, Renaud Sirdey, Oana Stan, Nicolas Quero, and Petr Kuznetsov. Randsolomon: optimally resilient multi-party random number generation protocol. *arXiv preprint*, 2021. [arXiv:2109.04911](https://arxiv.org/abs/2109.04911).
- 10 Big Dipper. Active validators. Accessed: 2022-03-15. URL: <https://cosmos.bigdipper.live/validators>.
- 11 C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):pp.288–323, 1988.
- 12 P. Ekparinya, V. Gramoli, and G. Jourjon. Impact of man-in-the-middle attacks on ethereum. In *Proc. 37th IEEE Int. Symp. Reliable Distrib. Syst. (SRDS)*, pages 11–20, October 2018.
- 13 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)*, February 2020.
- 14 The eth2 upgrades. Accessed: 2022-03-26. URL: <https://ethereum.org/en/eth2/>.
- 15 Ethereum. Shard chains. Accessed: 2022-03-15. URL: <https://ethereum.org/en/upgrades/shard-chains/>.
- 16 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. doi:10.1109/DSN48063.2020.00040.
- 17 Adem Efe Gencer, Robbert van Renesse, and Emin Gün Sirer. Short paper: Service-oriented sharding for blockchains. In Aggelos Kiayias, editor, *Financial Cryptography and Data Security*, pages 393–401, 2017.
- 18 Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles, SOSP '17*, pages 51–68, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3132747.3132757.
- 19 Pedro Herrera. Dapp industry report – january 2022. Accessed: 2022-03-10. URL: <https://dappradar.com/blog/dapp-industry-report-january-2022>.
- 20 Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ewa Syta, and Bryan Ford. Omniledger: A secure, scale-out, decentralized ledger via sharding. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 583–598, 2018. doi:10.1109/SP.2018.000–5.
- 21 Jae Kwon and Ethan Buchman. Cosmos white paper. Accessed: 2021-25-03. URL: <https://v1.cosmos.network/resources/whitepaper>.
- 22 Loi Luu, Viswesh Narayanan, Chaodong Zheng, Kunal Baweja, Seth Gilbert, and Prateek Saxena. A secure sharding protocol for open blockchains. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, CCS '16*, pages 17–30, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2976749.2978389.
- 23 Satoshi Nakamoto. Bitcoin: a peer-to-peer electronic cash system, 2008.
- 24 C. Natoli and V. Gramoli. The balance attack or why forkable blockchains are ill-suited for consortium. In *47th IEEE/IFIP Int. Conf. Dependable Syst. and Netw. (DSN)*, June 2017.

- 25 Cosmos Networks. Map of zones. Accessed: 2022-03-10. URL: <https://mapofzones.com/?testnet=false&period=24&tableOrderBy=ibcVolume&tableOrderSort=desc>.
- 26 randao.org. Randao: Verifiable random number generation. Technical report, randao.org, 2017. Accessed February 2022. URL: [https://www.randao.org/whitepaper/Randao\\_v0.85\\_en.pdf](https://www.randao.org/whitepaper/Randao_v0.85_en.pdf).
- 27 Team Rocket. Snowflake to avalanche: A novel metastable consensus protocol family for cryptocurrencies. Technical report, Avalanche Foundation, 2018. Accessed: 2021-12-01. URL: <https://ipfs.io/ipfs/QmUy4jh5mGNZvLkjies1RWM4YuvJh5o2FYopNPVYwrRVGV>.
- 28 Team Rocket, Maofan Yin, Kevin Sekniqi, Robbert van Renesse, and Emin Gün Sirer. Scalable and probabilistic leaderless bft consensus through metastability. Technical Report 1906.08936v2, arXiv, 2019. [arXiv:1906.08936v2](https://arxiv.org/abs/1906.08936).
- 29 The ZILLIQA Team. The zilliqa technical whitepaper. Technical report, Zilliqa, 2017. Accessed February 2022. URL: <https://docs.zilliqa.com/whitepaper.pdf>.
- 30 Deepal Tennakoon, Yiding Hua, and Vincent Gramoli. Collachain: A bft collaborative middleware for decentralized applications, 2022. [arXiv:2203.12323](https://arxiv.org/abs/2203.12323).
- 31 SJ Wels. Guaranteed-tx: The exploration of a guaranteed cross-shard transaction execution protocol for ethereum 2.0. Master’s thesis, University of Twente, 2019.
- 32 Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger, 2015. Yellow paper.
- 33 Mahdi Zamani, Mahnush Movahedi, and Mariana Raykova. Rapidchain: Scaling blockchain via full sharding. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS ’18*, pages 931–948, New York, NY, USA, 2018. Association for Computing Machinery. [doi:10.1145/3243734.3243853](https://doi.org/10.1145/3243734.3243853).