Setchain: Improving Blockchain Scalability with Byzantine Distributed Sets and Barriers

Margarita Capretto
IMDEA Software Institute, Pozuelo de Alarcón, Madrid, Spain

Martín Ceresa
IMDEA Software Institute, Pozuelo de Alarcón, Madrid, Spain

Antonio Fernández Anta
IMDEA Networks Institute, Leganés, Madrid, Spain

Antonio Russo
IMDEA Networks Institute, Leganés, Madrid, Spain

César Sánchez
IMDEA Software Institute, Pozuelo de Alarcón, Madrid, Spain

Abstract

Blockchain technologies are facing a scalability challenge, which must be overcome to guarantee a wider adoption of the technology. This scalability issue is due to the use of consensus algorithms to guarantee the total order of the chain of blocks (and of the transactions within each block). However, total order is often overkilling, since important advanced applications of smart-contracts do not require a total order among *all* operations. A much higher scalability can potentially be achieved if a more relaxed order (instead of a total order) can be exploited.

In this paper, we propose a distributed concurrent data type, called *Setchain*, which improves scalability significantly. A Setchain implements a *grow-only set object* whose elements are not ordered, unlike conventional blockchain operations. When convenient, the Setchain allows forcing a synchronization barrier that assigns permanently an epoch number to a subset of the latest elements added. Therefore, two operations in the same epoch are not ordered, while two operations in different epochs are ordered by their respective epoch number. We present different Byzantine-tolerant implementations of Setchain, prove their correctness and report on an empirical evaluation of a prototype implementation.

Our results show that Setchain is orders of magnitude faster than consensus-based ledgers, since it implements grow-only sets with epoch synchronization instead of total order. Moreover, since the Setchain barriers can be synchronized with the underlying blockchain, Setchain objects can be used as a *sidechain* to implement many smart contract solutions with much faster operations than on basic blockchains.

2012 ACM Subject Classification Security and privacy → Distributed systems security

Keywords and phrases Distributed systems, blockchain, byzantine distributed objects, consensus, Setchain.

1 Introduction

1.1 The Problem

Distributed ledgers (also known as *blockchains*) were first proposed by Nakamoto in 2009 [21] in the implementation of Bitcoin, as a method to eliminate trustable third parties in electronic payment systems. Modern blockchains incorporate smart contracts [28, 33], which are state-full programs stored in the blockchain that describe the functionality of the transactions, including the exchange of cryptocurrency. Smart contracts allow to describe sophisticated functionality, enabling many applications in decentralized finances (DeFi)[1], decentralized governance, Web3, etc.

The main element of all distributed ledgers is the "blockchain", which is a distributed object that contains, packed in blocks, the ordered list of transactions performed on behalf of the users [14, 13]. This object is maintained by multiple servers without a central authority by using consensus algorithms that are resilient to Byzantine attacks.

However, a current major obstacle for a faster widespread adoption of blockchain technologies is their limited scalability, due to the delay introduced by Byzantine consensus algorithms [8, 31]. Ethereum [33], one of the most popular blockchains, is limited to less than 4 blocks per minute, each containing less than two thousand transactions. Bitcoin [21] offers even lower throughput. These figures are orders of magnitude slower than what many decentralized applications require, and can ultimately jeopardize the adoption of the technology in many promising domains. This limit in the throughput of the blockchain also increases the price per operation, due to the high demand to execute operations.

Consequently, there is a growing interest in techniques to improve the scalability of blockchains [20, 35]. Approaches include (1) the search for faster consensus algorithms [32], (2) the use of parallel techniques, like sharding [10], (3) building application-specific blockchains with Inter-Blockchain Communication capabilities [34], [19], or (4) extracting functionality out of the blockchain, while trying to preserve the guarantees of the blockchain: the "layer 2" (L2) approach [17]. L2 approaches include the computation off-chain of Zero-Knowledge proofs [2], which only need to be checked on-chain (hopefully more efficiently) [1], the adoption of limited (but useful) functionality like *channels* (e.g., Lightning [22]), or the deployment of optimistic rollups (e.g., Arbitrum [18]) based on avoiding running the contracts in the servers (except when needed to annotate claims and resolve disputes).

In this paper, we propose an alternative approach to increase blockchain scalability that exploits the following observation. It has been traditionally assumed that cryptocurrencies require total order to guarantee the absence of double-spending. However, many useful applications and functionalities (including cryptocurrencies [16]) can tolerate more relaxed guarantees, where operations are only *partially ordered.* We propose here a Byzantine-fault tolerant implementation of a distributed grow-only set [27, 5], equipped with an additional operation for introducing points of barrier synchronization (where all servers agree on the contents of the set). Between barriers, elements of the distributed set can temporarily be known by some but not all servers. We call this distributed data structure a Setchain. A blockchain $\mathcal{B}$ implementing Setchain (as well as blocks) can align the consolidation of the blocks of $\mathcal{B}$ with barrier synchronizations, obtaining a very efficient set object as side data type, with the same Byzantine-tolerance guarantees that $\mathcal{B}$ itself offers.

There are two extreme implementations of a transaction set with epochs (like Setchain) in the context of blockchains:

1.1.0.1 A Completely off-chain implementation

The major drawback is that from the point of view of the underlying blockchain the resulting implementation does not have the trustability and accountability guarantees that blockchains offer. One example of this approach are *mempools.* Mempools (short for memory pools) are a P2P data type used by most blockchains to maintain a set of pending transactions.

[1] As of December 2021, the monetary value locked in DeFi was estimated to be around $100B, according to Statista `https://www.statista.com/statistics/1237821/defi-market-size-value-crypto-locked-usd/`.

Mempools fulfill two objectives: (1) to prevent distributed attacks to the servers that mine blocks and (2) to serve as a pool of transaction requests from where block producers select operations. Nowadays, mempools are receiving a lot of attention, since they suffer from lack of accountability and are a source of attacks [26, 25], including front-running [9, 24, 30]. Our proposed data structure, Setchain, offers a much stronger accountability, because it is resilient to Byzantine attacks and the contents of the set that Setchain maintains is public and cannot be forged.

1.1.0.2 Completely on-chain solution

Consider the following implementation (in a language similar to Solidity), where `add` is used to add elements, and `epochinc` to increase epochs.

```
contract Epoch {
  uint public epoch = 0;
  set public the_set = emptyset;
  mapping(uint => set) public history;
  function add(elem data) public {
    the_set.add(data);
  }
  function epochinc() public {
    history[++epoch] = the_set.setminus(history);
  }
}
```

Since `epoch`, `the_set`, and `history` are defined `public`, there is an implicit getter function for each of them[2]. One problem of this implementation is that every time we add an element, `the_set` gets bigger, which can affect the required cost to execute the contract. A second more important problem is that adding elements is *slow*—as slow as interacting with the blockchain—while our main goal is to provide a much faster data structure than the blockchain.

Our approach is faster, and can be deployed independently of the underlying blockchain or synchronized with the blockchain nodes. Thus, it lies between of these two extremes.

For any given blockchain $\mathcal{B}$, we propose an implementation of Setchain that (1) is much more efficient than implementing and executing operations directly in $\mathcal{B}$; (2) offers the same decentralized guarantees against Byzantine attacks than $\mathcal{B}$, and (3) can be synchronized with the evolution of $\mathcal{B}$, so contracts could potentially inspect the contents of the Setchain. In a nutshell, these goals are achieved by using faster operations for the coordination among the servers (namely, reliable broadcast) for non-synchronized element insertions, and use only a consensus like algorithm for epoch changes.

1.2 Motivation

The potential applications that motivate the development of Setchain include:

1.2.1 Mempool

User transaction requests are nowadays stored in a mempool before they are chosen by miners, and once mined the information is lost. Recording and studying the evolution of mempools would require an additional object serving as a mempool *log system*, which must

[2] In a public blockchain this function is not needed, since the set of elements can be directly obtained from the state of the blockchain.

be fast enough to record every attempt of interaction with the mempool without affecting the underlying blockchain's performance. Setchain as a sidechain can be used to implement one such trustable log system.

1.2.2 Scalability by L2 Optimistic Rollups

Optimistic rollups, like Arbitrum [18], use the fact that computation can be done outside the blockchain, posting on-chain only claims about its evolution. In this optimistic strategy users can propose the next state of the "contract." After some time, the arbitrer smart contract on-chain assumes that a given proposed step is correct, and executes the annotated effects. A conflict resolution algorithm, also part of the contract on-chain, is used to resolve disputes. This protocol does not require a strict total order, but only a record of the actions proposed. Moreover, conflict resolutions can be reduced to claim validations, which could be performed by the maintainers of the Setchain.

1.2.3 Sidechain Data

Finally, Setchain can also be used as a general side-chain service used to store and modify data synchronized with the blocks. Applications that require only to update information in the storage space of a smart contract, like digital registries, can benefit from faster (and therefore cheaper) methods to manipulate the storage without invoking expensive blockchain operations.

1.3 Contributions.

In summary, the contributions of the paper are the following:

- the design and implementation of a side-chain data structure called *distributed Setchain*,
- several implementations of Setchain, providing different levels of abstraction and algorithmic implementation improvements,
- an empirical evaluation of a prototype implementation, which suggests that Setchain is several orders of magnitude faster than consensus.

2 Preliminaries

In this section, we present the model of computation as well as the building blocks used in our Setchain algorithms.

2.1 Model of Computation

A distributed system consists of processes—clients and servers—with an underlying communication graph in which each process can communicate with every other process. The communication is performed using message passing. Each process computes independently and at its own speed, and the internals of each process remain unknown to other processes. Message transfer delays are arbitrary but finite and also remain always unknown to processes. The intention is that servers will communicate among themselves to implement a distributed data type with certain guarantees, and clients can communicate with servers to exercise the data type.

Processes can fail arbitrarily, but the number of failing servers is bounded by f, and the total number of servers, n, is at least $3f+1$. We assume *reliable channels* between non-Byzantine (correct) processes, so no message is lost, duplicated or modified. Each process (client or server) has a pair of public and private keys. The public keys have been distributed reliably to all the processes that may interact with each other. Therefore, we discard the possibility of spurious or fake processes. We assume that messages are authenticated, so that messages corrupted or fabricated by Byzantine processes are detected and discarded by correct processes [7]. As result, communication between correct processes is reliable but asynchronous by default. However, for the set consensus service we use as a basic building block, partial synchrony is required [6, 15], as presented below. Observe that this requirement is only for the messages and computation of the protocol implementing this service. Finally, we assume that there is a mechanism for clients to create "valid objects" that servers can check locally. In the context of blockchains this is implemented using public-key cryptography.

2.2 Building Blocks

We will use four building blocks to implement Setchain:

2.2.1 Byzantine Reliable Broadcast (BRB)

The BRB service [3, 23], allows to broadcast messages to a set of processes guaranteeing that messages sent by correct processes are eventually received by *all* correct processes and that all correct processes eventually receive *the same* set of messages. The service provides a primitive BRB.Broadcast(m) for sending messages and an event BRB.Deliver(m) for receiving messages. Some important properties of BRB are:

- **BRB-Validity:** If a correct process p_i executes BRB.Deliver(m) and m was sent by a correct process p_j, then p_j executed BRB.Broadcast(m) in the past.
- **BRB-Termination:** If a correct process p executes BRB.Broadcast(m), then all correct processes (including p) eventually execute BRB.Deliver(m).

Note that BRB does not guarantee the delivery of messages in the same order to two different correct participants.

2.2.2 Byzantine Atomic Broadcast (BAB)

The BAB service [11] extends BRB with an additional guarantee: a total order of delivery of the messages. BAB provides the same operation and event as BRB, which we will rename as BAB.Broadcast(m) and BAB.Deliver(m). In addition to validity and termination, BAB services also provide:

- **Total Order:** If two correct processes p and q both BAB.Deliver(m) and BAB.Deliver(m'), then p delivers m before m', if and only if q delivers m before m'.

BAB has been proven to be as hard as consensus [11], and thus, is subject to the same limitations [15].

2.2.3 Byzantine Distributed Grow-only Sets (DSO) [5]

Sets are one of the most basic and fundamental data structures in computer science, which typically include operations for adding and removing elements. Adding and removing operations do not commute, and thus, distributed implementations require additional mechanisms to keep replicas synchronized to prevent conflicting local states. One solution is to allow only additions. Hence, a grow-only set is a set in which elements can only be added but not removed (implementable as a conflict-free replicated data structure [27]).

Let A be an alphabet of values. A grow-only set GS is a concurrent object maintaining an internal set $GS.S \subseteq A$ offering two operations for any process p:

- GS.`add`(r) : adds an element $r \in A$ to the set $GS.S$.
- GS.`get`$()$: retrieves the internal set of elements $GS.S$.

Initially, the set $GS.S$ is empty. A Byzantine distributed grow-only set object (DSO) is a concurrent grow-only set implemented in a distributed manner [5] and tolerant to Byzantine attacks. Some important properties of these DSOs are:

- **Byzantine Completeness**: All `get`() and `add`() operations invoked by correct processes eventually complete.
- **DSO-AddGet**: All `add`(r) operations will eventually result in r being in the set returned by *all* `get`().
- **DSO-GetAdd**: Each element r returned by `get`() was added using `add`(r) in the past.

2.2.4 Set Byzantine Consensus (SBC)

SBC, introduced in RedBelly [6], is a Byzantine-tolerant distributed problem, similar to consensus. In SBC, each participant proposes a set of elements (in the particular case of RedBelly, a set of transactions). After SBC finishes, all correct servers agree on a set of valid elements which is guaranteed to be a subset of the union of the proposed sets. Intuitively, SBC efficiently runs binary consensus to agree on the sets proposed by each participant, such that if the outcome is positive then the set proposed is included in the final set consensus. Some properties of SBC are:

- **SBC-Termination**: every correct process eventually decides a set of elements.
- **SBC-Agreement**: no two correct process decide different sets of elements.
- **SBC-Validity**: when SBC is used on sets of transactions, the decided set of transactions is a valid non-conflicting subset of the union of the proposed sets.
- **SBC-Nontriviality**: if all processes are correct and propose an identical set, then this is the decided set.

The RedBelly algorithm [6] solves SBC in a system with partial synchrony: there is an unknown global stabilization time after which communication is synchronous. (Other SBC algorithms may have different partial synchrony assumptions.) Then, [6] proposes to use SBC to replace consensus algorithms in blockchains, seeking to improve scalability, because all transactions to be included in the next block can be decided with one execution of the SBC algorithm. Every server computes the same block by applying a deterministic function that totally orders the decided set of transactions, removing invalid or conflicting transactions.

Our use of SBC is different from implementing a blockchain. We use it to synchronize the barriers between local views of distributed grow-only sets. To guarantee that all elements are eventually assigned to epochs, we need the following property in the SBC service used.

- **SBC-Censorship-Resistance**: there is a time τ after which, if the proposed sets of all correct processes contain the same element e, then e will be in the decided set.

In RedBelly, this property holds because after the global stabilization time, all set consensus rounds decide sets from correct processes [6, Theorem 3].

3 The Setchain Distributed Data Structure

A key concept of Setchain is the *epoch* number, which is a global counter that the distributed data structure maintains. The synchronization barrier is realized as an epoch change: the epoch number is increased and the elements in the grow-only set that have not been assigned a previous epoch are stamped with the new epoch number.

3.1 API and Server State of the Setchain

We consider a universe U of elements that client processes can inject into the set. We also assume that servers can locally validate an element $e \in U$. A **Setchain** is a distributed data structure where a set of server nodes, $\mathbb{D}$, maintain:

- a set `the_set` $\subseteq U$ of elements added;
- a natural number `epoch` $\in \mathbb{N}$;
- a map `history` : $[1..\texttt{epoch}] \rightarrow \mathcal{P}(U)$, that describes the sets of elements that have been stamped with an epoch number ($\mathcal{P}(U)$ denotes the power set of U).

Each server node $v \in \mathbb{D}$ supports three operations, available to any client process:

- v.`add`(e): requests to add e to `the_set`.
- v.`get`(): returns the values of `the_set`, `history`, and `epoch`, as seen by v.
- v.`epoch_inc`(h) triggers an epoch change (i.e., a synchronization barrier). It must hold that $h = \texttt{epoch} + 1$.

Informally, a client process p invokes a v.`get`() operation in node v to obtain (S, H, h), which is v's view of set v.`the_set` and map v.`history`, with domain $[1 \ldots h]$. Process p invokes v.`add`(e) to insert a new element e in v.`the_set`, and v.`epoch_inc`($h+1$) to request an epoch increment. At server v, the set v.`the_set` contains the knowledge of v about elements that have been added, including those that have not been assigned an epoch yet, while v.`history` contains only those elements that have been assigned an epoch. A typical scenario is that an element $e \in U$ is first perceived by v to be in `the_set`, to eventually be stamped and copied to `history` in an epoch increment. However, as we will see, some implementations allow other ways to insert elements, in which v gets to know e for the first time during an epoch change. The operation `epoch_inc`() initiates the process of collecting elements in `the_set` at each node and collaboratively decide which ones are stamped with the current epoch.

Initially, both `the_set` and `history` are empty and `epoch` $= 0$ in every correct server. Note that client processes can insert elements to `the_set` through `add`(), but only servers decide how to update `history`, which client processes can only influence by invoking `epoch_inc`().

At a given point in time, the view of `the_set` may differ from server to server. The Setchain data structure we propose only provides eventual consistency guarantees, as defined next.

3.2 Desired Properties

We specify now properties of correct implementations of Setchain. We provide first a low-level specification that assumes that clients interact with a *correct* server. Even though clients cannot be sure of whether the server they contact is correct we will see how they can later check and confirm that the operations were successful. These low-level primitives are also used in Section 7 to build a protocol that allows correct clients to perform operations even when they interact with Byzantine servers, at the price of performance.

We start by requiring from a Setchain that every `add`, `get`, and `epoch_inc` operation issued on a correct server eventually terminates. We say that element e is in epoch i in history H (e.g., returned by a `get` invocation) if $e \in H(i)$. We say that element e is in H if there is an epoch i such that $e \in H(i)$. The first property states that epochs only contain elements coming from the grow-only set.

▶ **Property 1** (Consistent Sets). *Let $(S, H, h) = v.\texttt{get}()$ be the result of an invocation to a correct server v. Then, for each $i \leq h, H(i) \subseteq S$.*

The second property states that every element added to a correct server is eventually returned in all future gets issued on the same server.

▶ **Property 2** (Add-Get-Local). *Let* $v.\texttt{add}(e)$ *be an operation invoked to a correct server* v*. Then, eventually all invocations* $(S, H, h) = v.\texttt{get}()$ *satisfy* $e \in S$*.*

The next property states that elements present in a correct server are propagated to all correct servers.

▶ **Property 3** (Add-Get). *Let* v, w *be two correct servers, let* $e \in U$ *and let* $(S, H, h) = v.\texttt{get}()$*. If* $e \in S$*, then eventually all invocations* $(S', H', h') = w.\texttt{get}()$ *satisfy that* $e \in S'$*.*

We assume in the rest of the paper that at every point in time, there is a future instant at which `epoch_inc()` is invoked and completed. This is a reasonable assumption in any real practical scenario, since it can be easily guaranteed using timeouts. Then, the following property states that all elements added are eventually assigned an epoch.

▶ **Property 4** (Eventual-Get). *Let* v *be a correct server, let* $e \in U$ *and let* $(S, H, h) = v.\texttt{get}()$*. If* $e \in S$*, then eventually all invocations* $(S', H', h') = v.\texttt{get}()$ *satisfy that* $e \in H'$*.*

The previous three properties imply the following property.

▶ **Property 5** (Get-After-Add). *Let* $v.\texttt{add}(e)$ *be an operation invoked on a correct server* v *with* $e \in U$*. Then, eventually all invocations* $(S, H, h) = w.\texttt{get}()$ *satisfy that* $e \in H$*, for all correct servers* w*.*

An element can be in at most one epoch, and no element can be in two different epochs even if the history sets are obtained from `get` invocations to two different (correct) servers.

▶ **Property 6** (Unique Epoch). *Let* v *be a correct server,* $(S, H, h) = v.\texttt{get}()$*, and let* $i, i' \leq h$ *with* $i \neq i'$*. Then,* $H(i) \cap H(i') = \emptyset$*.*

All correct server processes agree on the epoch contents.

▶ **Property 7** (Consistent Gets). *Let* v, w *be correct servers, let* $(S, H, h) = v.\texttt{get}()$ *and* $(S', H', h') = w.\texttt{get}()$*, and let* $i \leq \min(h, h')$*. Then* $H(i) = H'(i)$*.*

Property 7 states that the histories returned by two `get` invocations to correct servers are one the prefix of the other. However, since two elements e and e' can be inserted at two different correct servers—which can take time to propagate—, the `the_set` part of `get` obtained from two correct servers may not be contained in one another.

Finally, we require that every element in the history comes from the result of a client adding the element.

▶ **Property 8** (Add-before-Get). *Let* v *be a correct server,* $(S, H, h) = v.\texttt{get}()$*, and* $e \in S$*. Then, there was an operation* $w.\texttt{add}(e)$ *in the past.*

Properties 1, 6, 7 and 8 are safety properties. Properties 2, 3, 4 and 5 are liveness properties.

4 Implementations

In this section, we describe implementations of Setchain that satisfy the properties in Section 3. We first describe a centralized sequential implementation, and then three distributed

Algorithm 0 Single server implementation.

```
Init: epoch ← 0,        history ← ∅
Init: the_set ← ∅
function Get( )
    return (the_set, history, epoch)
function Add(e)
    assert valid(e)
    the_set ← the_set ∪ {e}
function EpochInc(h)
    assert h ≡ epoch + 1
    proposal ← the_set \ ⋃_{k=1}^{epoch} history(k)
    history ← history ∪ {⟨h, proposal⟩}
    epoch ← epoch + 1
```

implementations. The first distributed implementation is built using a Byzantine distributed grow-only set object (DSO) to maintain `the_set`, and Byzantine atomic broadcast (BAB) for epoch increments. The second distributed implementation is also built using DSO, but it uses Byzantine reliable broadcast (BRB) to announce epoch increments and set Byzantine consensus (SBC) for epoch changes. Finally, the third one uses local sets, BRB for broadcasting elements and epoch increment announcements, and SBC for epoch changes.

4.1 Sequential Implementation

Alg. 0 shows a centralized solution, which maintains two local sets, `the_set`—to record added elements—, and `history`, which is implemented as a collection of pairs $\langle h, A\rangle$ where h is an epoch number and A is a set of elements. We use `history`(h) to refer to the set A in the pair $\langle h, A\rangle \in$ `history`. A natural number `epoch` is incremented each time there is a new epoch. The operations are: Add(e), which checks that element e is valid and adds it to `the_set`, and Get(), which returns (`the_set`, `history`, `epoch`).

4.2 Distributed Implementations

4.2.1 First approach. DSO and BAB

Alg. 1 uses two external services: DSO and BAB. We denote messages with the name of the message followed by its content as in "`epinc`$(h, \mathit{proposal}, i)$". The variable `the_set` is not a local set anymore, but a DSO initialized empty with Init() in line 2. The function Get() invokes the DSO Get() function (line 4) to fetch the set of elements. The function EpochInc(h) triggers the mechanism required to increment an epoch and reach a consensus on the elements. This process begins by computing a local *proposal* set, of those elements added but not stamped (line 14). The *proposal* set is then broadcasted using a BAB service alongside the epoch number h and the server node id i (line 15). Then, the server waits to receive exactly $2f + 1$ proposals, and keeps the set of elements E present in at least $f + 1$ proposals, which guarantees that each element $e \in E$ was proposed by at least one correct server. The use of BAB guarantees that every message sent by a correct server eventually reaches every other correct server in the *same order*, so all correct servers use the same set of $2f + 1$ proposals. Therefore, all correct servers arrive to the same conclusion, and the set E is added as epoch h in `history` in line 20.

Algorithm 1 Server i implementation using DSO and BAB

1: **Init:** $\texttt{epoch} \leftarrow 0, \quad \texttt{history} \leftarrow \emptyset$
2: **Init:** $\texttt{the_set} \leftarrow$DSO.Init()
3: **function** GET( )
4:     **return** $(\texttt{the\_set}.\text{Get}(), \texttt{history}, \texttt{epoch})$
5: **function** ADD(e)
6:     **assert** $valid(e)$
7:     $\texttt{the\_set}.\text{Add}(e)$
12: **function** EPOCHINC(h)
13:     **assert** $h \equiv \texttt{epoch} + 1$
14:     $proposal \leftarrow \texttt{the\_set}.\text{Get}() \setminus \bigcup_{k=1}^{\texttt{epoch}} \texttt{history}(k)$
15:     BAB.Broadcast($\texttt{epinc}(h, proposal, i)$)
16: **upon** (BAB.Deliver($\texttt{epinc}(h, proposal, j)$)
17:       from $2f + 1$ different servers j for the same h) **do**
18:     **assert** $h \equiv \texttt{epoch} + 1$
19:     $E \leftarrow \{e : e \in proposal$ for at least $f + 1$ different j$\}$
20:     $\texttt{history} \leftarrow \texttt{history} \cup \{\langle h, E \rangle\}$
21:     $\texttt{epoch} \leftarrow \texttt{epoch} + 1$
22: **end upon**

Alg. 1, while easy to understand and prove correct, is not efficient. To start, in order to complete an epoch increment, it requires at least $3f + 1$ calls to EpochInc(h) to different servers, so at least $2f + 1$ proposals are received (the f Byzantine severs may not propose anything). Another source of inefficiency comes from the use of off-the-shelf building blocks. For instance, every time a DSO Get() is invoked, many messages are exchanged to compute a reliable local view of the set [5]. Similarly, every epoch change requires a DSO Get() in line 14 to create a proposal. Additionally, line 17 requires waiting for $2f + 1$ atomic broadcast deliveries to take place. The most natural implementations of BAB services solve one consensus per message delivered (see Fig. 7 in [4]), which would make this algorithm very slow. We solve these problems in two alternative algorithms.

4.2.2 Second approach. Avoiding BAB

Alg. 2 improves the performance of Alg. 1 in several ways. First, it uses BRB to propagate epoch increments, so a client does not need to contact more than one server. Second, the use of BAB and the wait for the arrival of $2f + 1$ messages in line 17 of Alg. 1 is replaced by using a SBC algorithm, which allows solving several consensus instances simultaneously.

Ideally, when an EpochInc(h) is triggered unstampped elements in the local `the_set` of each correct server should be stamped with the new epoch number and added to the set `history`. However, we need to guarantee that for every epoch the set `history` is the same in every correct server. Alg 1 enforced this using BAB and counting sufficient received messages. Alg. 2 uses SBC to solve several independent consensus instances simultaneously, one on each participant's proposal. Line 14 broadcasts an invitation to an epoch change, which causes correct servers to build a proposed set and propose it the SBC. There is one instance of SBC per epoch change, identified by h. With SBC each correct server receives the same set of proposals (where each proposal is a set of elements). Then, every node applies the same function to the same set of proposals reaching the same conclusion on how to update `history`(h). The function preserves elements that are present in at least $f + 1$ proposed sets,

Algorithm 2 Server i implementation using DSO, and reliably broadcast (BRB) and set Byzantine consensus (SBC).

... ▷ Get and Add as in Alg. 1
function EpochInc(h)
  assert $h \equiv \texttt{epoch} + 1$
  BRB.Broadcast($\texttt{epinc}(h)$)
upon (BRB.Deliver($\texttt{epinc}(h)$) and $h < \texttt{epoch} + 1$) **do**
  drop
end upon
upon (BRB.Deliver(h) and $h \equiv \texttt{epoch} + 1$) **do**
  assert $\mathit{prop}[h] \equiv \mathit{null}$
  $\mathit{prop}[h] \leftarrow \texttt{the\_set}.\text{Get}() \setminus \bigcup_{k=1}^{\texttt{epoch}} \texttt{history}(k)$
  SBC[h].Propose($\mathit{prop}[h]$)
end upon
upon (SBC[h].SetDeliver($\mathit{propset}$) and $h \equiv \texttt{epoch} + 1$) **do**
  $E \leftarrow \{e : e \in \text{at least } f + 1 \text{ different } \mathit{propset}[j]\}$
  $\texttt{history} \leftarrow \texttt{history} \cup \{\langle h, E\rangle\}$
  $\texttt{epoch} \leftarrow \texttt{epoch} + 1$
end upon

which are guaranteed to have been proposed by some correct server. Observe that Alg. 2 still triggers one invocation of the DSO Get at each server to build the local proposal.

4.2.3 Final approach. BRB and SBC without DSOs

Alg. 3, avoids the cascade of messages that DSO Get calls require by dissecting the internals of the DSO, and incorporating the internal steps in the Setchain algorithm directly. This idea exploits the fact that *a correct* Setchain *server* is a *correct client* of the DSO, and there is no need for the DSO to be defensive (this illustrates that using Byzantine resilient building blocks does not compose efficiently, but exploring this general idea is out of the scope of this paper).

Alg. 3 implements `the_set` using a local set (line 2). Elements received in Add(e) are propagated using BRB. At any given point in time two correct servers may have a different local sets (due to pending BRB deliveries) but each element added in one server will eventually be known to all others. The local variable `history` is only updated in line 25 as a result of a SBC round. Therefore, all correct servers will agree on the same sets formed by unstamped elements proposed by some server. Additionally, Alg. 3 updates `the_set` to account for elements that are new to the server (line 26) , guaranteeing that all elements in `history` are also in `the_set`. Note that this opens the opportunity to add elements directly by proposing them during an epoch change without broadcasting them before. This optimization is exploited in Section 6 to speed up the algorithm further. As a final note, Alg. 3 allows a Byzantine server to bypass Add to propose elements, which will be accepted as long as the elements are valid. This is equivalent to a client proposing an element using an Add operation, which is then successfully propagated in an epoch change.

5 Proof of Correctness

We prove now the correctness of Alg. 3. We first show that all stamped elements are in `the_set`, which implies Prop. 1 (*Consistent Sets*).

Algorithm 3 Server implementation using a local set, Byzantine reliable broadcast (BRB) and set Byzantine consensus (SBC).

1: **Init:** $\texttt{epoch} \leftarrow 0$, $\texttt{history} \leftarrow \emptyset$
2: **Init:** $\texttt{the\_set} \leftarrow \emptyset$
3: **function** Get( )
4: **return** $(\texttt{the\_set}, \texttt{history}, \texttt{epoch})$
5: **function** Add(e)
6: **assert** $valid(e)$ and $e \notin \texttt{the\_set}$
7: BRB.Broadcast($\texttt{add}$(e))
8: **upon** (BRB.Deliver($\texttt{add}(e)$)) **do**
9: **assert** $valid(e)$
10: $\texttt{the\_set} \leftarrow \texttt{the\_set} \cup \{e\}$
11: **end upon**
12: **function** EpochInc(h)
13: **assert** $h \equiv \texttt{epoch} + 1$
14: BRB.Broadcast($\texttt{epinc}(h)$)
15: **upon** (BRB.Deliver($\texttt{epinc}(h)$) and $h < \texttt{epoch} + 1$) **do**
16: **drop**
17: **end upon**
18: **upon** (BRB.Deliver($\texttt{epinc}(h)$) and $h \equiv \texttt{epoch} + 1$) **do**
19: **assert** $prop[h] \equiv \emptyset$
20: $prop[h] \leftarrow \texttt{the\_set} \setminus \bigcup_{k=1}^{\texttt{epoch}} \texttt{history}(k)$
21: SBC[h].Propose($prop[h]$)
22: **end upon**
23: **upon** (SBC[h].SetDeliver($propset$) and $h \equiv \texttt{epoch} + 1$) **do**
24: $E \leftarrow \{e : e \in propset[j], valid(e) \wedge e \notin \texttt{history}\}$
25: $\texttt{history} \leftarrow \texttt{history} \cup \{\langle h, E \rangle\}$
26: $\texttt{the\_set} \leftarrow \texttt{the\_set} \cup E$
27: $\texttt{epoch} \leftarrow \texttt{epoch} + 1$
28: **end upon**

▶ **Lemma 1.** *For every correct server v, at the end of each function/upon, $\bigcup_h v.\texttt{history}(h) \subseteq v.\texttt{the\_set}$.*

Proof. Let v be a server. The only way to add elements to $v.\texttt{history}$ is at line 25, which is followed by line 26 which adds the same elements to $v.\texttt{the\_set}$. The only other instruction that modifies $v.\texttt{the\_set}$ is line 10 which only makes the set grow. ◀

▶ **Lemma 2.** *Let v be a correct server and e an element in $v.\texttt{the\_set}$. Then e will eventually be in $w.\texttt{the\_set}$ for every correct server w.*

Proof. Initially, $v.\texttt{the\_set}$ is empty. There are two ways to add an element e to $v.\texttt{the\_set}$: (1) At line 10, so e is valid and was received via a BRB.Deliver($\texttt{add}(e)$). By Properties **BRB-Validity** and **BRB-Termination** of BRB (see Section 2), every correct server w will eventually execute BRB.Deliver($\texttt{add}(e)$), and then (since e is valid), w will add it to $w.\texttt{the\_set}$ in line 10. (2) At line 26, so element e is valid and was received as an element in one of the sets in *propset* from SBC[h].SetDeliver(*propset*) with $h = v.\texttt{epoch} + 1$. By properties **SBC-Termination SBC-Agreement** and **SBC-Validity** of SBC (see Section 2), all correct servers agree on the same set of proposals. Therefore, if v adds e then w either adds it or has it already in its $w.\texttt{history}$ which implies by Lemma 1 that $e \in w.\texttt{the\_set}$. In either case, e will eventually be in $w.\texttt{the\_set}$. ◀

Lemma 2, and the code of Add() and line 4 of Get() imply Prop. 2 (*Add-Get-Local*) and Prop. 3 (*Add-Get*). The following lemmas reason about how elements are stamped.

▶ **Lemma 3.** *Let v be a correct server and $e \in v.\texttt{history}(h)$ for some h. Then, for any $h' \neq h$, $e \notin v.\texttt{history}(h')$.*

Proof. It follows directly from the check that e is not injected at v.`history`(h) if $e \in v$.`history` in line 25. ◀

▶ **Lemma 4.** *Let v and w be correct servers. At a point in time, let h be such that $v.\texttt{epoch} \geq h$ and $w.\texttt{epoch} \geq h$. Then $v.\texttt{history}(h) = w.\texttt{history}(h)$.*

Proof. The proof proceeds by induction on `epoch`. The base case is `epoch` $= 0$, which holds trivially since v.`history`$(0) = w$.`history`$(0) = \emptyset$. Variable `epoch` is only incremented in one unit in line 27, after `history`(h) has been changed in line 25 when $h =$ `epoch` $+ 1$. In that line, v and w are in the same phase on SBC (for the same h). By **SBC-Agreement**, v and w receive the same *propset*, both v and w validate all elements equally, and (by inductive hypothesis), for each $h' \leq$ `epoch` it holds that $e \in v$.`history`(h') if and only if $e \in w$.`history`(h'). Therefore, in line 25 both v and w update `history`(h) equally, and after line 27 it holds that v.`history(epoch)` $= w$.`history(epoch)`. ◀

▶ **Lemma 5.** *Let v and w be correct servers. If $e \in v.\texttt{the\_set}$. Then, eventually e is in $w.\texttt{history}$.*

Proof. By Lemma 2 every correct server w will satisfy $e \in w$.`the_set` at some $t > \tau$. By assumption, there is a new EpochInc() after t (let the epoch number be h). If e is already in `history`(h') for $h' < h$ we are done, since from Lemma 4 in this case at the end of the SBC phase for h' every correct server node w has e in w.`history`(h'). If e is not in `history` at t then, **SBC-Censorship-Resistance** guarantees that the decided set will contain e. Therefore, at line 25 every correct server w will add e to w.`history`(h). ◀

Lemmas 4 and 5 imply that all elements will be stamped, i.e. Prop. 3 (*Eventual-Get*). Prop. 5 follows from Prop. 3. Lemma 3 directly implies Prop. 6 (*Unique Epoch*). Finally, Lemma 4 is equivalent to Prop. 7 (*Consistent Gets*).

Finally, we discuss Prop. 8 (*Add-before-Get*). If valid elements can only be created by clients and added using Add(e) the property trivially holds. If valid elements can be created by, for example Byzantine servers, then they can inject elements in `the_set` and `history` of correct servers without using Add(). They can either execute directly a BRB.Broadcast or directly via the SBC in epoch rounds. In these case, Alg. 3 satisfies a weaker version of (*Add-before-Get*) that states that elements returned by Get() are either added by Add(), by a BRB.Broadcast or injected in the SBC phase.

6 Empirical Evaluation

We have implemented the server code for DSO, BRB and SBC and using these building blocks we have implemented Alg. 2 and Alg. 3. Our prototype is written in Golang [12] 1.16 with message passing using ZeroMQ [29] over TCP. Our testing platform uses Docker running on a server with 2 Intel Xeon CPU processors at 3GHz with 36 cores and 256GB RAM, running Ubuntu 18.04 Linux64. Each Setchain server node was wrapped in a Docker container with no limit on CPU or RAM usage. Alg. 2 implements a Setchain and a DSO as two standalone executables that communicate using remote procedure calls on the internal

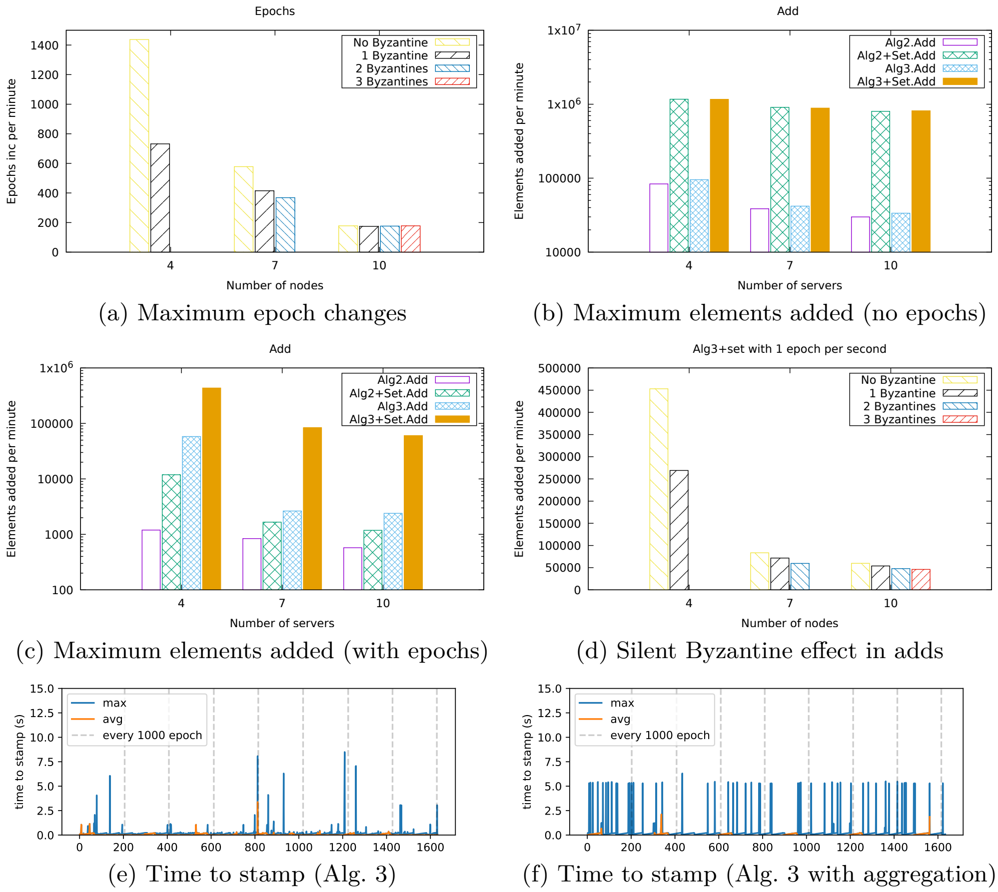

Figure 1 Experimental results. Alg. 2+set and Alg. 3+set are the versions of the algorithms with aggregation. Byzantine servers are simply silent.

loopback network interface of the Docker container. The RPC server and client are taken from the Golang standard library. For Alg. 3 everything resides in a single executable. For both algorithms, we evaluate two versions, one where each element inserted causes a broadcast and another where servers aggregate locally inserted elements until a maximum message size (of 10^6 elements) or a maximum element timeout (of 5s) is reached. In all cases elements have 116-126 bytes.

We evaluate empirically the following hypothesis:

- (H1): The maximum rate of elements that can be inserted is much higher than the maximum epoch rate.
- (H2): Alg. 3 performs better than Alg. 2.
- (H3): The aggregated versions perform better than the basic versions.
- (H4) Silent Byzantine servers do not affect dramatically the performance.
- (H5) The performance does not degrade over time.

To evaluate these hypotheses, we carried out the experiments described below and reported in Fig. 1. In all cases, operations are injected by clients running within the same Docker container. Resident memory was always enough such that in no experiment the operating system needed to recur to disk swapping. All the experiments consider deployments with 4, 7, or 10 server nodes, and each running experiment reported is taken from the average of 10 executions.

We tested first how many epochs per minute our Setchain implementations can handle. In these runs, we did not add any element and we incremented the epoch rate to find out the smallest latency between an epoch and the subsequent one. We run it with 4, 7, and 10 nodes, with and without Byzantines servers. This is reported in Fig. 1(a).

In our second experiment, we estimated empirically how many elements per minute can be added using our four different implementations of Setchain (Alg. 2 and Alg. 3 with and without aggregation), without any epoch increment. This is reported in Fig. 1(b). In this experiment Alg. 2 and Alg. 3 perform similarly. With aggregation Alg. 2 and Alg. 3 also perform similarly, but one order of magnitude better than without aggregation, confirming (H3). Putting together Fig. 1(a) and (b) one can conclude that sets are three orders of magnitude faster than epoch changes, confirming (H1).

In our third experiment, we compare the performance of our implementations combining epoch increments and insertion of elements. We set the epoch rate at 1 epoch change per second and calculated the maximum add ratio. The outcome is reported in Fig. 1(c), which shows that Alg. 3 outperforms Alg. 2. In fact, Alg. 3+set even outperforms Alg. 2+set by a factor of roughly 5 for 4 nodes and by a factor of roughly 2 for 7 and 10 nodes. Alg. 3+set can handle 8x the elements added by Alg. 3 for 4 nodes and 30x for 7 and 10 nodes. The benefits of Alg. 3+set over Alg. 3 increase as the number of nodes increase because Alg. 3+set avoids the broadcasting of elements which generates a number of messages that is quadratic in the number of nodes in the network. This experiment confirms (H2) and (H3). The difference between Alg. 3 and Alg. 2 was not observable in the previous experiment (without epoch changes) because the main difference is in how servers proceed to collect elements to vote during epoch changes.

The next experiment explores how silent Byzantine servers affect Alg. 3+set. We implement silent Byzantine servers and run for 4,7 and 10 nodes with an epoch change ratio of 1 per second, calculating the maximum add rate. This is reported in Fig. 1(d). Silent Byzantine servers degrade the speed for 4 nodes as in this case the implementation considers the silent server very frequently in the validation phase, but it can be observed that this effect is much smaller for larger number of servers, validating (H4).

In the final experiment, we run 4 servers for a long time (30 minutes) with an epoch ratio of 5 epochs per second and add requests to 50% of the maximum rate. We compute the time elapsed between the moment in which the client requests an add and the moment at which the element is stamped. Fig. 1(e) and (f) show the maximum and average times for the elements inserted in the last second. In the case of Alg. 3, the worst case during the 30 minutes experiment was around 8 seconds, but the majority of the elements were inserted within 1 sec or less. For Alg. 3+set the maximum times were 5 seconds repeated in many occasions during the long run (5 seconds was the timeout to force a broadcast). This happens when an element fails to be inserted using the set consensus and ends up being broadcasted. In both cases the behavior does not degrade with long runs, confirming (H5).

Considering that epoch changes is essentially a set consensus, our experiments suggest that inserting elements in a Setchain is three orders of magnitude faster than performing consensus. However, a full validation of this hypothesis would require to fully implement Setchain on performant gossip protocols and compare with comparable consensus implementations.

Algorithm 4 Correct client protocol for DPO (for Alg. 2 and 3).

function DPO.ADD(e)
  call Add(e) in $f+1$ different servers.
function DPO.GET( )
  call Get() in at least $3f+1$ different servers.
  wait $2f+1$ resp s.(`the_set`, `history`, `epoch`)
  $S \leftarrow \{e | e \in s.\texttt{the\_set} \text{ in at least } f+1 \text{ servers } s\}$
  $H \leftarrow \emptyset$
  $i \leftarrow 1$
  $N \leftarrow \{s : s.\texttt{epoch} \geq i\}$
  while $\exists E : |\{s \in N : s.\texttt{history}(i) = E\}| \geq f+1$ **do**
    $H \leftarrow H \cup \{\langle i, E \rangle\}$
    $N \leftarrow N \setminus \{s : s.\texttt{history}(i) \neq E\}$
    $N \leftarrow N \setminus \{s : s.\texttt{epoch} = i\}$
    $i \leftarrow i+1$
  return $(S, H, i-1)$
function DPO.EPOCHINC(h)
  call EpochInc(h) in $f+1$ different servers.

7 Distributed Partial Order Objects (DPO)

The algorithms presented in Section 4 and the proofs in Section 5 consider the case of clients contacting a correct server. Obviously, client processes do not know if they are contacting a Byzantine or correct process, so a client protocol is required to encapsulate the details of the distributed system. We describe now such a client protocol inspired by the one for DSO [5], which involves the exchange of several more messages than contacting a single server with a request. We later describe a more efficient "*try and check*" alternative.

The general idea of the client protocol is to interact with enough servers to guarantee that some are correct and ensure the desired behavior. The Setchain API has methods that wait for a result (Get) and methods that do not require a response (EpochInc and Add). Alg. 4 shows the client protocol. To guarantee contacting at least one correct server, we need to send $f+1$ messages. Note that each message may trigger different broadcasts.

The wrapper algorithm for function Get can be split in two parts. First, the protocol contacts $3f+1$ nodes, and waits for at least $2f+1$ responses (f Byzantine servers may refuse to respond). The response from server s is (s.`the_set`, s.`history`, s.`epoch`). The protocol then computes S as those elements known to be in `the_set` by at least $f+1$ servers (which includes at least one correct server). To compute H, the code goes incrementally epoch by epoch as long as at least $f+1$ servers within the set N (which is initialized with all the servers that responded with non-empty histories) agree on a set E of elements in epoch i. If $f+1$ servers agree that E is the set of elements in epoch i, this is indeed the case. We also remove from N those servers that either do not know more epochs or that incorrectly reported something different than E. Once this process ends, the sets S and H, and the latest processed epoch are returned. It is guaranteed that `history` $\subseteq$ `the_set`.

We also present an alternative faster optimistic client. In this approach correct servers sign cryptographically a hash of the set of elements in an epoch, and insert this hash in the Setchain as an element. Clients only perform **a single** Add(e) request to one server, hoping it will be a correct server. After waiting for some time, the client invokes a Get from **a single** server (which again can be Byzantine) and check whether e is in some epoch signed by (at least) $f+1$ servers, in which case the epoch is correct and e has been successfully inserted and stamped. Note that this requires only one message per Add and one message per Get.

8 Concluding Remarks

We presented a novel distributed data-type, called Setchain, that implements a grow-only set with epochs, and tolerates Byzantine server nodes. We provided a low-level specification of desirable properties of Setchains and presented three distributed implementations, where the most efficient one uses Byzantine Reliable Broadcast and RedBelly set Byzantine consensus. Our preliminary empirical evaluation suggests that the performance of inserting elements in Setchain is three orders of magnitude faster than with consensus.

Future work includes developing the motivating applications listed in the introduction, for example, mempool logs using Setchains, and L2 faster optimistic rollups. We will also study how to equip blockchains with Setchain (synchronizing blocks and epochs) to allow smart-contracts to access the Setchain. An important problem to solve is how clients of the Setchain pay for the usage (even if a much smaller fee than for the blockchain itself).

Setchain may be used to implement a solution to front-running. Mempools encode information about what it is about to happen in blockchains, so anyone observing them can predict the next operations to be mined, and take actions to their benefit. *Front-running* is the action of observed transaction request and maliciously inject transactions to be executed before the observed ones [9, 30] (by paying a higher fee to a miner). Setchain can be used to *detect* front-running since it can serve as a basic mechanism to build a mempool that is efficient and serves as a log of requests. Additionally, Setchains can be used as a building block to solve front-running where users encrypt their requests using a multi-signature decryption scheme, where participant decrypting servers decrypt requests after they are chosen for execution by miners once the order has already been fixed.

Our Setchain exploits a specific partial orders that relaxes the total order imposed by blockchains. As future we will explore other partial orders and their uses, for example, federations of Setchain, one Setchain per smart-contract, etc.

There are also interesting problems for foundational future work. Alg. 3 shows that Byzantine tolerant building blocks do not compose efficiently, because each building is pessimistic and does not exploit the fact that when building a correct sever, the client of the Byzantine tolerant building block is correct. Also, our analysis shows that Byzantine behavior of server nodes can be modeled by a collection of simple interactions with BRB and SBC, so it is possible to model all Byzantine behavior to simplify reasoning.

References

1. Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *Proc. of S&P'14*, pages 459–474, 2014. doi:10.1109/SP.2014.36.
2. Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Succinct non-interactive Zero Knowledge for a von Neumann architecture. In *Proc. of USENIX Sec.'14*, pages 781–796. USENIX, August 2014. URL: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/ben-sasson.
3. Gabriel Bracha. Asynchronous byzantine agreement protocols. *Inf. Comput.*, 75(2):130–143, 1987. doi:10.1016/0890-5401(87)90054-X.
4. Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, mar 1996. doi:10.1145/226643.226647.

5 Vicent Cholvi, Antonio Fernández Anta, Chryssis Georgiou, Nicolas Nicolaou, Michel Raynal, and Antonio Russo. Byzantine-tolerant distributed grow-only sets: Specification and applications. In *Proc. of FAB'21*, page 2:1–2:19, 2021.

6 Tyler Crain, Christopher Natoli, and Vincent Gramoli. Red belly: A secure, fair and scalable open blockchain. In *Proc. of S&P'21*, pages 466–483, 2021. doi:10.1109/SP40001.2021.00087.

7 F. Cristian, H. Aghili, R. Strong, and D. Volev. Atomic broadcast: from simple message diffusion to byzantine agreement. In *25th Int'l Symp. on Fault-Tolerant Computing*, pages 431–, 1995. doi:10.1109/FTCSH.1995.532668.

8 Kyle Croman, Christian Decker, Ittay Eyal, Adem Efe Gencer, Ari Juels, Ahmed Kosba, Andrew Miller, Prateek Saxena, Elaine Shi, Emin Gün Sirer, Dawn Song, and Roger Wattenhofer. On scaling decentralized blockchains. In *Financial Crypto. and Data Security*, pages 106–125. Springer, 2016.

9 Philip Daian, Steven Goldfeder, T. Kell, Yunqi Li, X. Zhao, Iddo Bentov, Lorenz Breidenbach, and A. Juels. Flash boys 2.0: Frontrunning in decentralized exchanges, miner extractable value, and consensus instability. *Proc. of S&P'20*, pages 910–927, 2020.

10 Hung Dang, Tien Tuan Anh Dinh, Dumitrel Loghin, Ee-Chien Chang, Qian Lin, and Beng Chin Ooi. Towards scaling blockchain systems via sharding. In *Proc. of SIGMOD'19*, pages 123—-140. ACM, 2019. doi:10.1145/3299869.3319889.

11 Xavier Défago, André Schiper, and Péter Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, 36(4):372–421, dec 2004. doi:10.1145/1041680.1041682.

12 Alan A.A. Donovan and Brian W. Kernighan. *The Go Programming Language.* Adison-Wesley, 2015.

13 Antonio Fernández Anta, Chryssis Georgiou, Maurice Herlihy, and Maria Potop-Butucaru. *Principles of Blockchain Systems.* Morgan & Claypool Publishers, 2021.

14 Antonio Fernández Anta, Kishori Konwar, Chryssis Georgiou, and Nicolas Nicolaou. Formalizing and implementing distributed ledger objects. *ACM Sigact News*, 49(2):58–76, 2018.

15 Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *JACM*, 32(2):374–382, 1985. doi:10.1145/3149.214121.

16 Rachid Guerraoui, Petr Kuznetsov, Matteo Monti, Matej Pavlovic, and Dragos-Adrian Seredinschi. The consensus number of a cryptocurrency. In *Proc. of PODC'19*, pages 307–316. ACM, 2019. doi:10.1145/3293611.3331589.

17 Maxim Jourenko, Kanta Kurazumi, Mario Larangeira, and Keisuke Tanaka. Sok: A taxonomy for layer-2 scalability related protocols for cryptocurrencies. *IACR Cryptol. ePrint Arch.*, 2019:352, 2019.

18 Harry Kalodner, Steven Goldfeder, Xiaoqi Chen, S. Matthew Weinberg, and Edward W. Felten. Arbitrum: Scalable, private smart contracts. In *27th USENIX Security Symposium*, pages 1353–1370. USENIX Assoc., 2018. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/kalodner.

19 Jae Kwon and Ethan Buchman. Cosmos whitepaper, 2019.

20 Zamani Mahdi, Mahnush Movahedi, and Mariana Raykova. Rapidchain: Scaling blockchain via full sharding. In *Proc. of CSS'18*, pages 931—-948. ACM, 2018. doi:10.1145/3243734.3243853.

21 Satoshi Nakamoto. Bitcoin: a peer-to-peer electronic cash system, 2009.

22 Joseph Poon and Thaddeus Dryja. The bitcoin lightning network: Scalable off-chain instant payments, 2016. URL: https://lightning.network/lightning-network-paper.pdf.

23 Michel Raynal. *Fault-Tolerant Message-Passing Distributed Systems: An Algorithmic Approach.* 01 2018. doi:10.1007/978-3-319-94141-7.

24 Robinson, Dan and Konstantopoulos, Georgios. Ethereum is a dark forest, 2020. URL: https://medium.com/@danrobinson/ethereum-is-a-dark-forest-ecc5f0505dff.

25 Muhammad Saad, Laurent Njilla, Charles Kamhoua, Joongheon Kim, DaeHun Nyang, and Aziz Mohaisen. Mempool optimization for defending against DDoS attacks in PoW-based blockchain systems. In *Proc. of ICBC'19*, pages 285–292, 2019. `doi:10.1109/BLOC.2019.8751476`.

26 Muhammad Saad, My T. Thai, and Aziz Mohaisen. POSTER: Deterring DDoS attacks on blockchain-based cryptocurrencies through mempool optimization. In *Proc. of ASIACCS'18*, pages 809—811. ACM, 2018. `doi:10.1145/3196494.3201584`.

27 Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Convergent and Commutative Replicated Data Types. *Bulletin- European Association for Theoretical Computer Science*, (104):67–88, June 2011. URL: `https://hal.inria.fr/hal-00932833`.

28 Nick Szabo. Smart contracts: Building blocks for digital markets. *Extropy*, 16, 1996.

29 The ZeroMQ authors. Zeromq, 2021. https://zeromq.org. URL: `https://zeromq.org`.

30 Christof Ferreira Torres, Ramiro Camino, and Radu State. Frontrunner jones and the raiders of the Dark Forest: An empirical study of frontrunning on the Ethereum blockchain. In *Proc of USENIX Sec.'21*, pages 1343–1359, 2021. URL: `https://www.usenix.org/conference/usenixsecurity21/presentation/torres`.

31 Shobha Tyagi and Madhumita Kathuria. *Study on Blockchain Scalability Solutions*, page 394–401. ACM, 2021. URL: `https://doi.org/10.1145/3474124.3474184`.

32 Ke Wang and Hyong S. Kim. Fastchain: Scaling blockchain system with informed neighbor selection. In *Proc. of IEEE Blockchain'19*, pages 376–383, 2019. `doi:10.1109/Blockchain.2019.00058`.

33 Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151:1–32, 2014.

34 Gavin Wood. Polkadot: Vision for a heterogeneous multi-chain framework. *White Paper*, 21, 2016.

35 Cheng Xu, Ce Zhang, Jianliang Xu, and Jian Pei. Slimchain: Scaling blockchain transactions through off-chain storage and parallel processing. *Proc. VLDB Endow.*, 14(11):2314–2326, jul 2021. `doi:10.14778/3476249.3476283`.