Improving Blockchain Scalability with the Setchain Data-type

MARGARITA CAPRETTO, IMDEA Software Institute, Spain and Universidad Politécnica de Madrid, Spain
MARTÍN CERESA, IMDEA Software Institute, Spain
ANTONIO FERNÁNDEZ ANTA, IMDEA Networks Institute, Spain
ANTONIO RUSSO, IMDEA Networks Institute, Spain and Universidad Carlos III de Madrid, Spain
CÉSAR SÁNCHEZ, IMDEA Software Institute, Spain

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 not fully necessary, 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 novel distributed concurrent data type, *Setchain*, which significantly improves scalability. A Setchain implements a *grow-only set* 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, agreed by consensus. 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.

Since the Setchain barriers can be synchronized with the underlying blockchain, Setchain objects can be used as a *sidechain* to implement many decentralized solutions with much faster operations than direct implementations on top of blockchains.

Finally, we also present an algorithm that encompasses into a single process the combined behavior of the Byzantine servers, which simplifies correctness proofs by encoding the general attacker in a concrete implementation.

CCS Concepts: • **Theory of computation → Distributed algorithms**; **Logic and verification**; • **Computing methodologies → Distributed algorithms**; • **Security and privacy → Formal methods and theory of security**.

Additional Key Words and Phrases: Distributed systems, blockchain, Byzantine distributed objects, consensus, Setchain.

Authors' addresses: Margarita Capretto, margarita.capretto@imdea.org, IMDEA Software Institute, Campus de Montegancedo s/n, 28223, Pozuelo de Alarcón, Madrid, Spain and Universidad Politécnica de Madrid, C. de los Ciruelos, 28660, Boadilla del Monte, Madrid, Spain; Martín Ceresa, martin.ceresa@imdea.org, IMDEA Software Institute, Campus de Montegancedo s/n, 28223, Pozuelo de Alarcón, Madrid, Spain; Antonio Fernández Anta, antonio.fernandez@imdea.org, IMDEA Networks Institute, Av. Mar Mediterráneo, 22, 28918, Leganés, Madrid, Spain; Antonio Russo, antonio.russo@imdea.org, IMDEA Networks Institute, Av. Mar Mediterráneo, 22, 28918, Leganés, Madrid, Spain and Universidad Carlos III de Madrid, Av. de la Universidad, 30, 28911, Leganés, Madrid, Spain; César Sánchez, cesar.sanchez@imdea.org, IMDEA Software Institute, Campus de Montegancedo s/n, 28223, Pozuelo de Alarcón, Madrid, Spain.

XXXX-XXXX/2025/3-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

ACM Reference Format:
Margarita Capretto, Martín Ceresa, Antonio Fernández Anta, Antonio Russo, and César Sánchez. 2025. Improving Blockchain Scalability with the Setchain Data-type. 1, 1 (March 2025), 28 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

1.1 The Problem

Distributed ledgers (also known as *blockchains*) were first proposed by Nakamoto in 2009 [22] in the implementation of Bitcoin, as a method to eliminate trustable third parties in electronic payment systems. Modern blockchains incorporate smart contracts [29, 34], which are immutable state-full programs stored in the blockchain that describe functionality of 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 totally ordered list of transactions performed on behalf of the users [14, 15]. The Blockchain object is maintained by multiple servers without a central authority using consensus algorithms that are resilient to Byzantine attacks.

A current major obstacle for a faster widespread adoption of blockchain technologies is their limited scalability, due to the limited throughput inherent to Byzantine consensus algorithms [9, 32]. Ethereum [34], one of the most popular blockchains, is limited to less than 4 blocks per minute, each containing less than two thousand transactions. Bitcoin [22] 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 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 [21, 36]. Approaches include: developing faster consensus algorithms [33]; implementing parallel techniques, like sharding [11]; application-specific blockchains with Inter-Blockchain Communication capabilities [20, 35]; executing smart contracts off-chain with the minimal required synchronization to preserve the guarantees of the blockchain— known as a "layer 2" (L2) approaches [18]. Different L2 approaches are (1) the off-chain computation of Zero-Knowledge proofs [2], which only need to be checked on-chain (hopefully more efficiently) [1], (2) the adoption of limited (but useful) functionality like *channels* (e.g., Lightning [23]), or (3) the deployment of optimistic rollups (e.g., Arbitrum [19]) 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 some uses of cryptocurrencies [17]) 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 [6, 28], 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 be temporarily known by some but not all servers. We call this distributed data structure 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

[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/.

very efficient set object as side data type, with the same Byzantine-tolerance guarantees that $\mathcal{B}$ itself offers.

Two extreme implementations of sets with epochs in the context of blockchains are:

A Completely off-chain implementation. The major drawback of having a completely off-chain implementation 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 the set of pending transactions. 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, 27], including front-running [10, 25, 31]. Our proposed data structure, Setchain, offers a much stronger accountability, because it is resilient to Byzantine attacks and the elements of the set that Setchain maintains are public and cannot be forged.

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);
  }
}
```

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 and synchronized with the blockchain nodes. Thus, Setchain lies between the two extremes described above.

For a 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 for non-synchronized element insertions, and using only consensus style algorithms for epoch changes.

1.2 Applications of Setchain

The potential applications that motivate the development of Setchain include:

1.2.1 Mempool. Most blockchains store transaction requests from users in a "mempool" before they are chosen by miners, and once mined the information from the mempool is lost. Recording and studying the evolution of mempools would require an additional object serving as a reliable mempool *log system*, which must be fast enough to record every attempt of interaction with the mempool without affecting the performance of the blockchain. Setchain can server as such trustable log system, in this case requiring no synchronization between epochs and blocks.

1.2.2 Scalability by L2 Optimistic Rollups. Optimistic rollups, like Arbitrum [19], exploit the fact that computation can be performed outside the blockchain, posting on-chain only claims about the effects of the transactions. In this manner Arbitrum maintainers propose the next state reached after executing several transactions. After some time, an arbitrator smart contract that is installed on-chain assumes that a proposed step is correct because the state has not been challenged, 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 resolution can be reduced to claim validation, which could be performed by the maintainers of the Setchain, removing the need for arbitration.

1.2.3 Sidechain Data. Finally, Setchain can also be used as a generic side-chain service used to store and modify data in a manner that is 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 having 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 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;
- a client protocol that describes how Setchain can be used as a distributed object which requires good clients to contact several servers for adding elements and for obtaining a correct view of the Setchain;
- a protocol that describes a much more efficient Setchain optimistic service that requires clients to contact only one server both for addition and for obtaining a correct state;
- a reduction from the combined behavior of several Byzantine servers to a single non-deterministic process that simplifies reasoning about the combined distributed system.

The rest of the paper is organized as follows. Section 2 contains preliminary model and assumptions. Section 3 describes the intended properties of Setchain. Section 4 describes three different implementations of Setchain where we follow an incremental approach. Alg. Basic and Slow are required to explain how we have arrived to Alg. Fast, which is the fastest and most robust. Section 5 proves the correctness of our three algorithms. Section 6 discusses an empirical evaluation of our prototype implementations of the different algorithms. Section 7 shows client protocols to correctly use Setchain under the presence of Byzantine servers. Section 8 presents a non-deterministic algorithm that simulates Byzantine behaviour. Finally, Section 9 concludes the paper.

2 PRELIMINARIES

We present now the model of computation and 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 network with 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 remains unknown to other processes. Message transfer delays are arbitrary but finite and also remain unknown to processes. The intention is that servers 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 (Byzantine) 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. Public keys were 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 messages corrupted or fabricated by Byzantine processes are detected and discarded by correct processes [8]. As result, communication between correct processes is reliable but asynchronous by default. However, the set consensus service we use as a basic building block requires partial synchrony [7, 16] (see Section 2.2.4). Partial synchrony is only required for messages and computations of the protocol implementing set consensus. 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 use four building blocks to implement Setchain:

2.2.1 Byzantine Reliable Broadcast (BRB). BRB services [3, 24] allow to broadcast messages to a set of processes guaranteeing that messages sent by correct processes are eventually received by *all* correct processes and all correct processes eventually receive *the same* set of messages. A BRB service provides a primitive BRB.Broadcast(m) for sending messages and an event BRB.Deliver(m) for receiving messages. We list the relevant properties of BRB required to prove properties of Setchain (Section 5):

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

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

2.2.2 Byzantine Atomic Broadcast (BAB). BAB services [12] extend BRB with an additional guarantee: a total order of delivery of the messages. BAB services provide the same operation and event as BRB, which we rename as BAB.Broadcast(m) and BAB.Deliver(m). However, in addition to validity and termination, BAB services also provide:

- **Total Order:** If two correct processes p and q both execute 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 [12], and thus, is subject to the same limitations [16].

2.2.3 Byzantine Distributed Grow-only Sets (DSO) [6]. 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, which is implementable as a conflict-free replicated data structure [28].

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 tolerant to Byzantine attacks [6]. We list the properties relevant to Setchain (Section 5):

- **Byzantine Completeness**: All get() and add(r) 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 [7], 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. We list the properties relevant to Setchain (Section 5):

- **SBC-Termination**: every correct process eventually decides a set of elements.
- **SBC-Agreement**: no two correct processes decide different sets of elements.
- **SBC-Validity**: the decided set of transactions is a 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 [7] 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, [7] 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. In RedBelly 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 [7, Theorem 3].

3 THE SETCHAIN DISTRIBUTED DATA STRUCTURE

The main contrivution of this paper is Setchain, a distributed Byzantine-fault tolerant data structure, that implementing an efficient grow-only set together with synchronization barriers. 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 to a previous epoch are stamped with the new epoch number.

3.1 The Way of Setchain

Before presenting the API of the Setchain, we reason about the expected path an element should travel in the Setchain and the properties we want the structure to have. The main goal of the Setchain side-chain data structure is to exploit the efficiency opportunity of the lack of order within a set using a fast grow-only set so users can add records (uninterpreted data), and thus, have evidence that such records exist. Either periodically or intentionally, users trigger an epoch change that creates an evidence of membership or a clear separation in the evolution of the Setchain. Therefore, the Setchain offers three methods: `add`, `get` and `epoch_inc`.

In a centralized implementation, we expect to find inserted elements immediately after issuing an `add`. In other words, after an `add`(e) we expect that the element e is in the result of `get`. In a distributed setting, such a restriction is too strong, and we instead expect elements to eventually be in the set. Additionally, implementations have to guarantee consistency between different correct nodes, i.e. they cannot contradict each other.

3.2 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 collection 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..epoch] $\rightarrow \mathcal{P}(U)$ describing 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 perceived by v[2];
- v.`epoch_inc`(h) triggers an epoch change (i.e. a synchronization barrier) if h = `epoch` + 1.

Informally, a client process p invokes a v.`get`() operation on 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. 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. Client processes can insert elements to `the_set` through operation `add`, but only servers decide how to update `history`, which client processes can only influence by invoking operation `epoch_inc`.

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

[2]In practice, we would have other query operations since values returned by `get()` operation may grow large.

3.3 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 are contacting is correct, we use these properties in Section 7 to build two correct clients: a pessimistic client contacting many servers to guarantee that sufficiently many are correct, and an optimistic client contacting only one server (hoping it will be a correct one) and can later check whether the operation was successful.

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

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

Algorithm Central Single server implementation.

Init: $\texttt{epoch} \leftarrow 0,\qquad \texttt{history} \leftarrow \emptyset$
Init: $\texttt{the\_set} \leftarrow \emptyset$
function GET( )
  return $(\texttt{the\_set}, \texttt{history}, \texttt{epoch})$
function ADD(e)
  assert $valid(e)$
  $\texttt{the\_set} \leftarrow \texttt{the\_set} \cup \{e\}$
function EPOCHINC(h)
  assert $h \equiv \texttt{epoch} + 1$
  $proposal \leftarrow \texttt{the\_set} \setminus \bigcup_{k=1}^{\texttt{epoch}} \texttt{history}(k)$
  $\texttt{history} \leftarrow \texttt{history} \cup \{\langle h, proposal\rangle\}$
  $\texttt{epoch} \leftarrow \texttt{epoch} + 1$

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 in some server w.*

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 defined in Section 3. We describe a centralized sequential implementation to build up intuition and three distributed 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 we replace BAB with Byzantine reliable broadcast (BRB) to announce epoch increments and set Byzantine consensus (SBC) for epoch changes. Finally, we replace DSO with local sets, use BRB for broadcasting elements and epoch increment announcements, and SBC for epoch changes, resulting in the fastest implementation.

4.1 Sequential Implementation

Alg. Central shows a solution, which maintains two local sets, `the_set`—to record added elements—and `history`, which keeps a collection of pairs $\langle h, A\rangle$ where h is an epoch number and A is a set of elements. We use $\texttt{history}(h)$ to refer to the set A in the pair $\langle h, A\rangle \in \texttt{history}$. 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 $(\texttt{the\_set}, \texttt{history}, \texttt{epoch})$.

There is only one way to add elements, through the use of operation Add. Since Alg. Central does not maintain a distributed data structure, but a centralized one, there are no Byzantine nodes. Therefore, every time clients interact with the (only) correct server.

The following implementations are distributed, and thus, they must incorporate some mechanism to prevent Byzantine nodes from manipulating the Setchain.

Algorithm Basic Server i implementation using DSO and BAB

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

4.2 Distributed Implementations

4.2.1 First approach. DSO and BAB. Alg. Basic uses two external services: DSO and BAB. We denote messages with the name of the message followed by its content as in "epinc(h, *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 belonging to epoch h. The consensus process begins by computing a local *proposal* set, of those elements added but not stamped (line 10). The *proposal* set is then broadcasted using the BAB service alongside the epoch number h and the server node id i (line 11). Then, server i 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 have the same set of $2f+1$ proposals. Therefore, all correct servers arrive at the same conclusion and the set E is added as epoch h in history in line 16.

Alg. Basic, while easy to understand and prove correct, is not efficient. First, 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 [6]. Similarly, every epoch change requires a DSO Get in line 10 to create a proposal. Additionally, line 13 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 [5]), which makes Alg. Basic very slow. We solve these problems in two alternative algorithms.

4.2.2 Second approach. Avoiding BAB. Alg. Slow improves the performance of Alg. Basic as follows. First, it uses BRB to propagate epoch increments. Second, the use of BAB and wait for the arrival of

Algorithm Slow Server i implementation using DSO, BRB and SBC.

```
...                                                    ▷ Get and Add as in Alg. Basic
function EpochInc(h)
    assert h ≡ epoch + 1
    BRB.Broadcast(epinc(h))
upon (BRB.Deliver(epinc(h)) and h < epoch + 1) do
    drop
upon (BRB.Deliver(epinc(h)) and h ≡ epoch + 1) do
    assert prop[h] ≡ null
    prop[h] ← the_set.Get() \ ⋃_{k=1}^{epoch} history(k)
    SBC[h].Propose(prop[h])
upon (SBC[h].SetDeliver(propset) and h ≡ epoch + 1) do
    E ← {e : e ∈ propset[j], valid(e) ∧ e ∉ history}
    the_set ← the_set.Add(E)
    history ← history ∪ {⟨h, E⟩}
    epoch ← epoch + 1
```

Algorithm Fast Server implementation using a local set, BRB and SBC.

```
Init: epoch ← 0,        history ← ∅
Init: the_set ← ∅
function Get( )
    return (the_set, history, epoch)
function Add(e)
    assert valid(e) and e ∉ the_set
    BRB.Broadcast(add(e))
upon (BRB.Deliver(add(e))) do
    assert valid(e)
    the_set ← the_set ∪ {e}
function EpochInc(h)
    assert h ≡ epoch + 1
    BRB.Broadcast(epinc(h))
upon (BRB.Deliver(epinc(h)) and h < epoch + 1) do
    drop
upon (BRB.Deliver(epinc(h)) and h ≡ epoch + 1) do
    assert prop[h] ≡ ∅
    prop[h] ← the_set \ ⋃_{k=1}^{epoch} history(k)
    SBC[h].Propose(prop[h])
upon (SBC[h].SetDeliver(propset) and h ≡ epoch + 1) do
    E ← {e : e ∈ propset[j], valid(e) ∧ e ∉ history}
    the_set ← the_set ∪ E
    history ← history ∪ {⟨h, E⟩}
    epoch ← epoch + 1
```

$2f + 1$ messages in line 13 of Alg. Basic is replaced by using a SBC algorithm, which allows solving several consensus instances simultaneously.

Ideally, when an EpochInc 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. Basic enforces this using BAB and counting sufficient received messages. Alg. Slow uses SBC to solve several independent consensus instances simultaneously, one on each participant's proposal. Line 10 broadcasts an invitation to an epoch change, which causes correct servers to build a proposed set and propose this set using the SBC. There is one instance of SBC per epoch change h, identified by SBC$[h]$. The SBC service guarantees that each correct server decides 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 $\texttt{history}(h)$. The function preserves elements that are valid and unstampped. 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 even further. As a final note, Alg. Slow allows a Byzantine server to bypass operation Add to propose elements, which will be accepted as long as the elements are valid. This is equivalent to clients proposing elements using operation Add, which are then successfully propagated in epoch changes. Alg. Slow still triggers one invocation of the DSO operation Get at each server to build the local proposal.

4.2.3 Final approach. BRB and SBC without DSOs. Alg. Fast avoids the cascade of messages that DSO operation 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 observation shows that using Byzantine resilient building blocks do not compose efficiently, but exploring this general idea is out of the scope of this paper.

Alg. Fast 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 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 23 as a result of a SBC round. Therefore, all correct servers will agree on the same sets formed by unstamped elements proposed by some servers. Additionally, Alg. Fast updates `the_set` to account for elements that are new to the server (line 22) , guaranteeing that all elements in `history` are also in `the_set`.

5 PROOF OF CORRECTNESS

We prove the correctness of the distributed algorithms presented in Section 4 with regard to the desired properties introduced in Section 3.

We first introduce lemmas to reason about how elements are stamped. These lemmas directly imply Property 4 (*Eventual-Get*), Property 6 (*Unique Epoch*) and Property 7 (*Consistent Gets*), respectively. We prove that our algorithms satisfy these lemmas in the following subsections.

Lemma 1. *Let v and w be correct servers. If $e \in v.\texttt{the\_set}$. Then, eventually e is in $w.\texttt{history}$.*

Lemma 2. *Let v be a correct server and h, h' two different epoch numbers. If $e \in v.\texttt{history}(h)$ then $e \notin v.\texttt{history}(h')$.*

Lemma 3. *Let v and w be correct servers. Let h be such that $h \leq v.\mathsf{epoch}$ and $h \leq w.\mathsf{epoch}$. Then $v.\texttt{history}(h) = w.\texttt{history}(h)$.*

It is easy to see that Property 5 (*Get-After-Add*) follows directly from Properties 2, 3 and 4.

5.1 Correctness of Alg. Basic

Property 1 (*Consistent Sets*) holds for Alg. Basic as the response of operation Get (line 4) is the tuple $(\texttt{the\_set.get}()cup\texttt{history}, \texttt{history}, \texttt{epoch}))$. Property 2 (*Add-Get-Local*) follows directly from Property **DSO-AddGet**, as this property verses about how elements are added to `the_set` implemented as a DSO.

Before proving that Alg. Basic satisfies the remaining properties, we prove the following auxiliary Lemma 4, which states that all elements stamped were proposed by a correct server.

LEMMA 4. *Let v be a correct server and e an element such that $e \in v.\mathtt{history}(h)$. Then, a correct server proposed e to be included in epoch h.*

PROOF. Let v be a correct server and e an element in the history set of epoch h of server v. The only point where the set $v.\mathtt{history}$ is updated is at line 16, where $v.\mathtt{history}(h)$ is defined to contain exactly the elements that were proposed for epoch h by $f+1$ different servers. Therefore, given that there are at most f Byzantine servers, at least one correct server proposed a set containing e for epoch h. □

We show that every element present in correct servers comes from the result of a client adding the element, which implies Property 8 (*Add-before-Get*)

LEMMA 5. *Let v be a correct server and e an element such that $e \in v.\mathtt{the\_set.get}() \cup v.\mathtt{history}$. Then, for some server w, operation $w.\mathtt{add}(e)$ was issued in the past.*

PROOF. Let v be a correct server and e an element such that $e \in v.\mathtt{the\_set.get}() \cup v.\mathtt{history}$. We split the proof into two cases, depending on whether the element e is in the set $v.\mathtt{the\_set}$ or in $v.\mathtt{history}$. If e is in $v.\mathtt{the\_set}$, then Property **DSO-GetAdd** guarantees that e was added using $w.\mathtt{add}(e)$ in the past. On the other hand, if element e is in $v.\mathtt{history}$, then by Lemma 4, e was proposed by a correct server. Since correct servers take elements to propose from `the_set`, by Property **DSO-GetAdd**, it follows that elements stamped were previously added by clients. In both cases, there was an operation $w.\mathtt{add}(e)$ in the past. □

The combination of Property 8 (*Add-before-Get*) and Property **DSO-AddGet** implies that elements present in a correct server are propagated to all correct servers, which is equivalent to Property 3 (*Get-Global*).

PROPOSITION 5.1. *Lemmas 1, 2 and 3 hold for Alg. Basic.*

We prove each lemma separately.

Proof of Lemma 3.

PROOF. We show that every two correct servers agree on the contents of epochs. Let v and w be two correct servers and h an epoch already processed by both. Since v and w are correct servers that computed $\mathtt{history}(h)$ (line 16), both servers v and w received $2f+1$ different BAB.Deliver messages proposing elements for epoch h. Properties **BAB-Termination(Global)** and **Total Order** guarantee that these are the same messages for both servers. Both v and w filter elements in the same way, by just keeping elements proposed by at least $f+1$ servers. Therefore, in line 16 both v and w update $\mathtt{history}(h)$ with the same elements. Hence, Lemma 3 holds for Alg. Basic. □

Proof of Lemma 1.

PROOF. Let e be an element in the set `the_set` of a correct server. It follows from properties **DSO-AddGet** and **DSO-GetAdd** that there is a point in time t after which e is in the set returned by all $\mathtt{the\_set.get}()$ in all correct servers. We assume that there is always eventually a new epoch increment, in particular, there is a new EpochInc(h) after t. If e is already part of the history, i.e. element e already has been assigned an epoch, there is nothing to do. Otherwise, by Lemma 3, e can not be in the history set of any correct server for all previous epochs. Then, when computing the proposal for epoch h (line 10) all correct servers will include e in their set. To compute the epoch h, a correct server w waits until it receives $2f+1$ BAB.Deliver messages of the form $\texttt{epinc}(h, proposal, j)$ from different servers (see line 13), and thus, server w collects at least $f+1$ messages from correct servers. Therefore, at least $f+1$ of the proposals received contain e, and thus, server w includes e in its set $\texttt{history}(h)$. This shows Lemma 1 for Alg. Basic. □

Proof of Lemma 2.

Proof. The proof proceeds by contradiction. Let v be a correct server and e an element such that element e belongs to two different epochs in v. Let h and h' be two different epochs such that $e \in v.\texttt{history}(h')$ and $e \in v.\texttt{history}(h)$. Moreover, without losing generality, assume $h < h'$. By Lemma 4, there is a correct server w that proposed element e to be included in epoch h'. Server w computed its proposed set of epoch h' (see line 10) as the set $\texttt{the\_set}.\text{Get}() \setminus \bigcup_{k=1}^{h'-1} \texttt{history}(k)$. However, by Lemma 3 and v, w correct server, both v and w have the same epoch h, i.e. $w.\texttt{history}(h) = v.\texttt{history}(h)$, and thus, $e \in \bigcup_{k=1}^{h'-1} \texttt{history}(k)$ as $h < h'$. Therefore, e can not be in set $\texttt{the\_set}.\text{Get}() \setminus \bigcup_{k=1}^{h'-1} \texttt{history}(k)$, meaning that e was not proposed by w for epoch h'. This contradiction follows from assuming that element e belongs to two different epochs in a correct server. Then, Lemma 2 holds for for Alg. Basic. □

5.2 Correctness of Alg. Slow

Alg. Slow also satisfies Property 1 (*Consistent Sets*), Property 2 (*Add-Get-Local*), and Property 3 (*Get-Global*). The first two properties are showed following the same reasoning used for Alg. Basic. Property 3 (*Get-Global*) follows from properties **DSO-AddGet** and **DSO-GetAdd**, which ensure that elements added to `the_set` of correct servers are eventually added to `the_set` of all correct servers and from Lemma 3 and Property **SBC-Termination**, which imply that elements stamped in correct servers will eventually be stamped in all correct servers.

Next, we prove for Alg. Slow lemmas 1, 2 and 3 introduced at the beginning of this section reasoning about how elements are stamped.

Proof of Lemma 1.

Proof. Elements added in correct servers will eventually be stamped in all correct servers. The proof is analogous to the proof for Alg. Basic above, but instead of relying on enough messages being BAB.Deliver, we rely on **SBC-Censorship-Resistance** guarantying element e is in the decided set. □

Proof of Lemma 2.

Proof. Lemma 2 for Alg. Slow follows directly from the fact that e is not added to a new epoch if it already belongs to the history of correct servers (see line 18). □

Proof of Lemma 3.

Proof. Let v and w be two correct servers. We show that v and w agree on the prefix of the `history` they both computed. The proof proceeds by induction on the epoch number $\texttt{epoch}$. The base case is $\texttt{epoch} = 0$, which holds since the history set in both correct servers is empty. The inductive hypothesis is that both servers, v and w, agree on the history up to epoch $\texttt{epoch}$, we show that both of them compute the same epoch next. Variable $\texttt{epoch}$ is only incremented by one in line 21, only after $\texttt{history}(\texttt{epoch}+1)$ has been changed in line 20. In that line, servers v and w are in the same phase on SBC (for the same h). By **SBC-Agreement**, servers v and w receive the same *propset*, both servers validate all elements and keep the same elements, because the history is the same up to epoch. Therefore, in line 20, both servers v and w compute the same $\texttt{history}(\texttt{epoch}+1)$ and, after line 21, both servers computed the same epoch, i.e. $v.\texttt{history}(\texttt{epoch}+1) = w.\texttt{history}(\texttt{epoch}+1)$. Hence, Lemma 3 holds for Alg. Slow. □

Finally, Alg. Slow does not satisfy Property 8 (*Add-before-Get*) as stated, so we prove a weaker version that states that elements returned by operation Get are either added by operation Add, by a the_set.Add, or injected during a set Byzantine consensus phase. Again, this can be seen from the code and the use of SBC as a building block. However, elements can only be created by clients, and thus, although Byzantine processes can injects elements, they can only inject elements that valid clients wanted to inject in first place.

5.3 Correctness of Alg. Fast

We show that Alg. Fast is correct. Property 2 (*Add-Get-Local*) follows directly from the code of function Add, line 4 of function Get and Property **BRB-Termination(Local)** of BRB. Moreover, all stamped elements are in the_set, which implies Property 1 (*Consistent Sets*).

Lemma 6. *For every correct server, the local set history is a subset of its local set* the_set *at the end of each procedure of Alg. Fast.*

Proof. Let v be a correct server. The only way to add elements to $v.\texttt{history}$ is at line 23 preceded by line 22 adding the same elements to $v.\texttt{the\_set}$. The other instruction that modifies $v.\texttt{the\_set}$ is line 10 which only makes the set grow. □

The following lemma states that elements in correct servers are eventually propagated to all correct servers, which is equivalent to Property 3 (*Get-Global*).

Lemma 7. *Let v be a correct server and e an element in $v.\texttt{the\_set}$. Then e will eventually be in the set* the_set *of every server.*

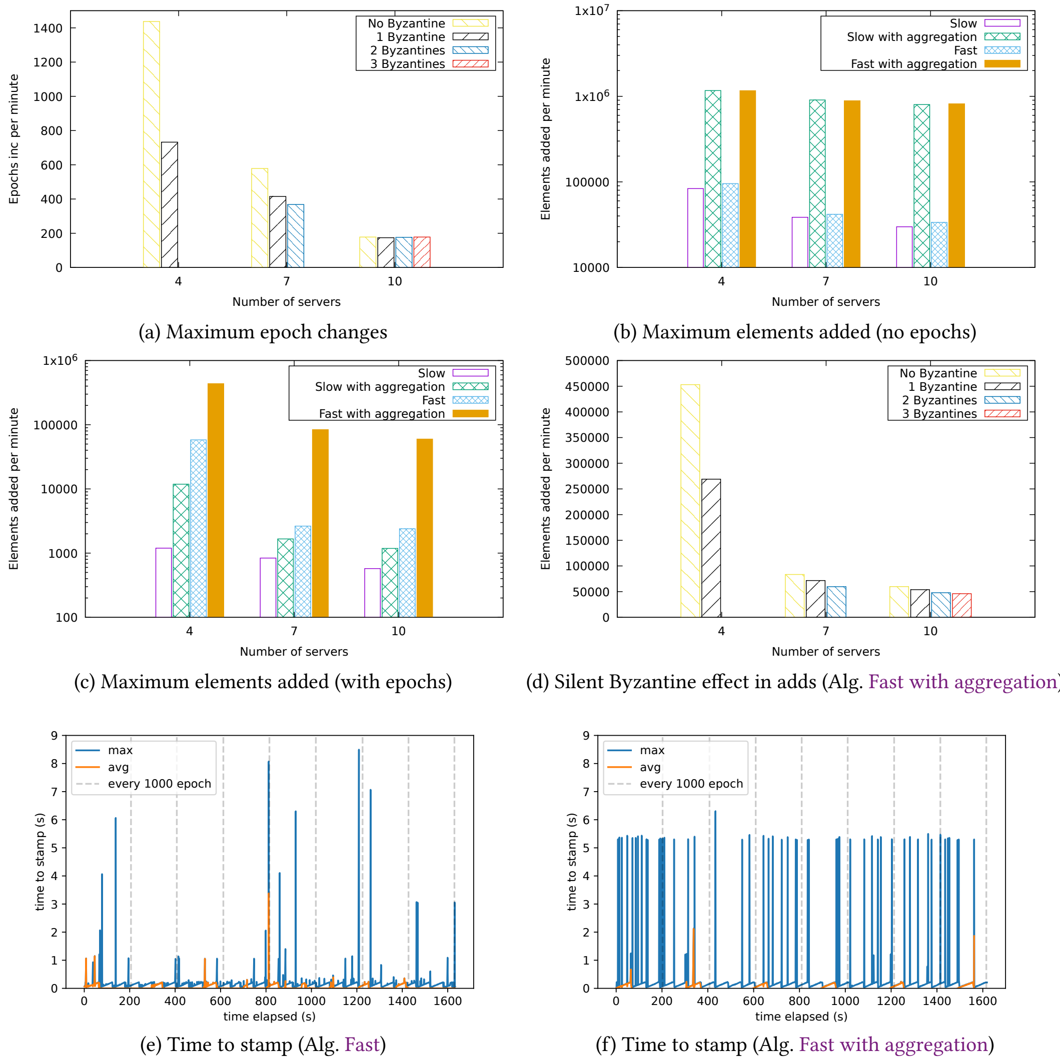
Proof. Initially, the set $v.\texttt{the\_set}$ is empty. There are two ways to add an element e to $v.\texttt{the\_set}$:

(1) At line 10, so element e is valid and was received via a BRB.Deliver(add(e)). By Property **BRB-Termination (Global)**, every correct server w will eventually execute BRB.Deliver(add(e)), and then (since e is valid), w will add it to $w.\texttt{the\_set}$ in line 10.
(2) At line 22, 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** and **SBC-Agreement**, all correct servers agree on the same set of proposals. Then, every correct server w will also eventually receive SBC[h].SetDeliver(*propset*). Therefore, if v adds e then w either adds it or has it already in its $w.\texttt{history}$ which implies by Lemma 6 that $e \in w.\texttt{the\_set}$.

In either case, e will eventually be in $w.\texttt{the\_set}$. □

Lemmas 1, 2 and 3 introduced at the beginning of the section are proved for Alg. Fast following a similar reasoning used for Alg. Slow, but replacing properties **DSO-AddGet** and **DSO-GetAdd** by Property 3 when proving Lemma 1.

Regarding Property 8 (*Add-before-Get*), Alg. Fast suffers from the same limitation as Alg. Slow because Byzantine servers can inject valid elements. Alg. Fast also satisfies a weaker version of (*Add-before-Get*) that states that elements returned by function Get were either added by function Add, by a BRB.Broadcast, or injected during a set Byzantine consensus phase. This is supported by our assumption that valid elements can only be created by clients.



(a) Maximum epoch changes

(b) Maximum elements added (no epochs)

(c) Maximum elements added (with epochs)

(d) Silent Byzantine effect in adds (Alg. Fast with aggregation)

(e) Time to stamp (Alg. Fast)

(f) Time to stamp (Alg. Fast with aggregation)

Fig. 1. Experimental results. Alg. Slow with aggregation and Alg. Fast with aggregation are the versions of the algorithms with aggregation. Byzantine servers are simply silent.

6 EMPIRICAL EVALUATION

We implemented the server code of Alg. Slow and Alg. Fast using our implementations of DSO, BRB and SBC. [3]

Our prototype is written in Golang [13] 1.16 with message passing style using ZeroMQ [30] over TCP. Our testing platform used Docker running on a server with 2 Intel Xeon CPU processors at 3GHz with 36 cores and 256GB RAM, running Ubuntu 18.04 Linux−64. Each Setchain server was packed in a Docker container with no limit on CPU or RAM usage. Alg. Slow implements Setchain and DSO as two standalone executables that communicate using remote procedure calls on the internal loopback network interface of the Docker container. The RPC server and client are taken from the Golang standard library. Alg. Fast resides in a single executable. We evaluated two

[3]The code is available open-source at https://github.com/imdea-software/setchain-basic

Algorithm Fast with aggregation Server implementation using a local set, BRB and SBC.

Init: epoch $\leftarrow 0$, history $\leftarrow \emptyset$
Init: the_set $\leftarrow \emptyset$, to_broadcast $\leftarrow \emptyset$
function GET( )
  return (the_set, history, epoch)
function ADD(e)
  assert $valid(e)$ and $e \notin$ the_set
  to_broadcast $\leftarrow$ to_broadcast $\cup \{e\}$
upon (BRB.Deliver(add(s))) **do**
  assert $valid(s)$
  the_set $\leftarrow$ the_set $\cup\ s$
  to_broadcast $\leftarrow$ to_broadcast $\setminus s$
… ▷ EpochInc and BRB.Deliver(epinc(h)) as in Alg. Fast
upon (SBC[h].SetDeliver($propset$) and $h \equiv$ epoch $+ 1$) **do**
  $E \leftarrow \{e : e \in propset[j], valid(e) \wedge e \notin$ history$\}$
  history $\leftarrow$ history $\cup \{\langle h, E\rangle\}$
  the_set $\leftarrow$ the_set $\cup\ E$
  to_broadcast $\leftarrow$ to_broadcast $\setminus E$
when ($|$to_broadcast$| > 1000000$ or to_broadcast.$oldest > 5s$) **do**
  BRB.Broadcast(add(to_broadcast))
  to_broadcast $\leftarrow \emptyset$

versions of each algorithm, one where each element insertion causes a broadcast and another where servers aggregate locally the elements inserted until a maximum message size (of 10^6 elements) or a maximum element timeout (of 5s) is reached. Elements have a size of 116-126 bytes in all cases. Alg. Fast with aggregation implements the aggregated version of Alg. Fast.

We evaluated empirically the following hypotheses:

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

To evaluate hypotheses H1 to H5, we carried out the experiments described below 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 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 silent Byzantines servers. The outcome 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. Slow and Alg. Fast with and without aggregation), without any epoch increment. This is reported in Fig. 1(b). In this experiment, Alg. Slow and Alg. Fast perform similarly. With aggregation Alg. Slow and Alg. Fast also perform similarly, but one order of magnitude better than the same algorithm without aggregation, confirming (H3).

Fig. 1(a) and (b) together suggest that sets are three orders of magnitude faster than epoch changes, confirming (H1).

The third experiment compares 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 ratio of Add operations. The outcome is reported in Fig. 1(c), which shows that Alg. Fast outperforms Alg. Slow. In fact, Alg. Fast even outperforms Alg. Slow with aggregation by a factor of roughly 5 for 4 nodes and by a factor of roughly 2 for 7 and 10 nodes. Alg. Fast with aggregation can handle 8 times the elements added by Alg. Fast for 4 nodes and 30 times for 7 and 10 nodes. The benefits of Alg. Fast with aggregation over Alg. Fast increase as the number of nodes increase because Alg. Fast with aggregation avoids 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. Fast and Alg. Slow 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. Fast with aggregation. We implement silent Byzantine servers and run for 4, 7 and 10 nodes with an epoch change ratio of 1 epoch 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 checks upon 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 clients request an add and the moment at which elements are stamped. Fig. 1(e) and (f) show the maximum and average times for elements inserted in the last second. In the case of Alg. Fast, the worst case during the 30 minutes experiment was around 8 seconds, but the majority of elements were inserted within 1 sec or less. For Alg. Fast with aggregation 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 are essentially 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 implement Setchain on performant gossip protocols and compare it with similar consensus implementations, which is left as future work.

7 CLIENT PROTOCOLS

Client protocols encapsulate the details of the distributed system to the clients. All properties described in Section 5 assume clients contact correct servers, but the implementations in Section 4 do not provide any guarantee to clients about whether the server their are contacting is correct. Therefore, clients cannot know if they are interacting with a Byzantine or a correct server.

In this section we describe two client protocols to interact with Setchains to ensure the correct exercise of its interface. First, we present a client protocol inspired by the DSO clients in [6], where clients contact several servers per operation. Later, we present a more efficient "optimistic" solution, based on try-and-check, that requires a simple change in the Setchain algorithms.

Algorithm 5 Correct client protocol for DPO (for Alg. Slow and Fast).

function DPO.$\textsc{Add}(e)$
  call $\text{Add}(e)$ in $f+1$ different servers.
function DPO.$\textsc{Get}(\,)$
  call Get() in at least $3f+1$ different servers.
  wait $2f+1$ responses s.(`the_set`, `history`, `epoch`)
  $S \leftarrow \{e \mid e \in s.\texttt{the\_set} \text{ in at least } f+1 \text{ servers } s\}$
  $H \leftarrow \emptyset$
  $i \leftarrow 1$
  $N \leftarrow \{s : s.\texttt{epoch} \geq i\}$
  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.$\textsc{EpochInc}(h)$
  call $\text{EpochInc}(h)$ in $f+1$ different servers.

7.1 Setchain as a Distributed Partial Order Object (DPO)

Alg. 5 shows the first client protocol. Intuitively, clients interact with a sufficient number of servers to guarantee that enough servers perform the desired operation correctly [6]. In functions Add and EpochInc, clients send $f+1$ requests to different servers, which gurantees that at least one of them is a correct server. Each request to a correct server trigger a BRB.Broadcast producing a cascade of messages that is quadratic on the number of servers.

Function Get begins by contacting $3f+1$ Setchain servers and waits for at least $2f+1$ responses (f Byzantine servers may refuse to respond). Each response is a triple of (`the_set`, `history`, `epoch`). The set `the_set` is computed as the set of elements known to be in the sets `the_set` of at least $f+1$ servers, which includes at least one correct server answer. To compute the history, the code proceeds incrementally epoch by epoch, stopping at the first epoch i for which less than $f+1$ servers agree on the set of elements. Note that if $f+1$ servers agree on the set of elements in epoch i, this set is indeed the set at epoch i. Clients also remove from the list of servers those servers that either do not know an epoch (either slow processes or Byzantine servers) or that disagree with at least $f+1$ servers. Once this process ends, the protocol returns the set `the_set`, the history, and the last epoch computed.

7.2 A Fast Optimistic Client

In this second approach, we modify correct servers to sign, using cryptographic signatures, each epoch number along with the hash of the set of elements of that epoch. This signature is inserted in the Setchain itself as an element, as shown in Alg. 6. We assume that the hash function is deterministic given a set of elements, so this ensures that all correct servers compute the same hash for each epoch. It follows that if enough signatures are collected, one can perform a local check and confirm the correctness of an epoch.

The new elements added by the servers do no produce a significant overhead in the Setchain. The number of new elements added to the Setchain per epoch is linear in the number of servers, and each epoch contains orders of magnitudes more elements than the number of servers.

Alg. 7 shows the optimistic client protocol. To insert an element e to the Setchain, the optimistic client performs **a single** $\text{Add}(e)$ request to one server hoping that such server is correct. After

Algorithm 6 Extend Alg. Fast adding the new epoch's hash cryptographically signed.

... ▷ previous lines as in Alg. Fast
upon (SBC[h].SetDeliver($propset$) and $h \equiv$ epoch + 1) **do**
  $E \leftarrow \{e : e \in propset[j], valid(e) \land e \notin \texttt{history}\}$
  $\texttt{history} \leftarrow \texttt{history} \cup \{\langle h, E\rangle\}$
  $\texttt{the\_set} \leftarrow \texttt{the\_set} \cup E$
  epoch $\leftarrow$ epoch + 1
  Add($epoch\_signature(h, sign(\langle h, hash(E)\rangle))$)

Algorithm 7 Optimistic client protocol for DPO (for Alg. 6).

function AddAndCheck(e)
  call Add(e) in 1 server.
  wait Δ_g
  call Get() in 1 server.
  wait resp (the_set, history, epoch)
  if $\exists E, i : \texttt{history}(i) = E \land e \in E \land (h, hash(E))$
    signed by $f + 1$ different servers is in the_set **then**
      return OK.
  else
      return Fail.

waiting for some time, the optimistic client invokes a Get from **a single** server (which again can be correct or Byzantine) and checks whether element e is in some epoch whose hash is signed by (at least) $f + 1$ different servers. Note that receiving one history in which element e is in an epoch is not enough to guarantee that e has been added to the setchain, since the server that provided history can be Byzantine and lie. However, cryptographic signatures cannot be forged, and thus, if $f + 1$ servers sign the hash of an epoch, this means that at least one correct server certifies the content of such an epoch.

7.3 Comparisson between clients

The two clients implemented in this section exploit a trade-off between latency and throughput. Optimistic clients may experience higher latency because after adding an element, they need to wait to check if the element has been inserted or retry the process. However, in the case optimistic clients contact a correct server, they only require one message per Add and one message per Get, dramatically reducing the number of messages exchanged.

Optimistic clients open the door to a whole class of optimizations, where one may ask what the best strategy for clients to get information from the Setchain is. Studying this question requires to reason about the probability of interacting with correct servers and determining the optimal frequency at which optimistic (or maybe other) clients need to contact different servers when not obtaining the desired outcome. In our setting, the worst Byzantine behavior is to hide information that guarantees that an element is in the Setchain. A systematic study of the best strategies and the trade-off involved is out of the scope of this work.

8 MODELLING BYZANTINE BEHAVIOR

In this section we introduce a non-deterministic process (see Alg. 8) that abstracts the combined behavior of all Byzantine processes (Alg. Fast). Formally proving properties[4] of Byzantine tolerant

[4]Proving here refers to rigorous machine reproducible or checkable proofs.

distributed algorithms is a very challenging task. Part of the difficulty comes from describing precisely what Byzantine processes can do. Our technique reduces the standard scenario with $n - f$ correct servers and f Byzantine servers into an *equivalent* scenario with $n - f$ correct servers and *one non-deterministic* server. This allows us to leverage many recent techniques for formally proving properties of (non-Byzantine) distributed algorithms. Our non-deterministic process abstracts away the behaviour of all Byzantine processes combined, even for Byzantine processes that enjoy instantaneous communication among themselves to coordinate attacks.

Byzantine processes share information between them. Setchain assumes that Byzantine processes can not forge valid elements (Section 2.1). Byzantine servers only become aware of the existence of valid elements when correct servers communicate these elements or when they are inserted by clients. We assume that as soon as a Byzantine process receives a valid element all other Byzantine processes know that element too. To model the information Byzantine processes can gather from the network, we add a new primitive SBC$[h]$.Inform(*prop*) exposing information from the set Byzantine consensus protocol. This primitive is triggered when servers call SBC$[h]$.Propose(*prop*) and satisfies the following property:

- **SBC-Inform-Validity:** if a process executes SBC$[h]$.Inform(*prop*) then some other process executed SBC$[h]$.Propose(*prop*) in the past.

Byzantine servers can discover elements proposed by correct servers during set consensus, before they are assigned an epoch. Moreover, Byzantine servers can return those elements in response to a Get, because they may be aware of their existence. To facilitate formal verification we have added the primitive SBC$[h]$.Inform in the model of computation to denote this potential knowledge.

Each possible action of Byzantine processes is modeled with the following non-deterministic functions: *havoc_subset*, *havoc_partition*, *havoc_element*, *havoc_number*, and *havoc_invalid_elems*. These functions generate, respectively, a random subset, element and partition from a given set; a random number, and random invalid elements. We do not focus on the semantics of these functions, we just use them to model Byzantine processes producing arbitrary sets of elements taken from a set of known values.

We model the collective behaviour of all Byzantine processes in Alg. 8. Alg. 8 maintains a local set *knowledge* to record all valid elements that the collective "Byzantine" process we model is aware of. This process exposes the same interface as Alg. Fast with an additional function Start. The function Start is invoked when the process starts and non-deterministically emits messages at arbitrary times (lines 21-27), using BRB and SBC primitives, as these are the only messages that are processed by correct servers. These messages can contain valid or invalid elements, but valid elements must be already known to the non-deterministic process. Similarly, when clients invoke function Get, Alg. 8 returns $(s_v \cup s_i, partition(s'_v \cup s'_i), h)$, where s_v and s'_v are sets of valid elements from *knowledge* while s_i and s'_i are sets of invalid elements (lines 2-4). Upon receiving a message, the non-deterministic process annotates all newly discovered valid elements in its local set *knowledge*.

We show that any execution of Setchain maintained by n servers implementing Alg. Fast out of which at most $1 \leq f < n/3$ are Byzantine can be mapped to an execution of Setchain maintained by $n - f$ **correct** servers implementing Alg. Fast and **one** server implementing Alg. 8 and vice versa.

Events. We represent with the following *events* the different interactions clients and servers can have with a Setchain plus the internal events of the Setchain reaching consensus.

- `get`() represents the invocation of function Get(),
- `add`(e) represents the invocation of function Add(e),
- BRB.Broadcast(x) represents the broadcast of `add` or `epinc` messages through the network, with x = `add`(e) or x = `epinc`(h) respectively,
- BRB.Deliver(x) represents the reception of message x,

Algorithm 8 Collective Byzantine Behaviour

Init: knowledge $\leftarrow \emptyset$
function Get( )
  return $(havoc\_subset(\texttt{knowledge} \cup generate\_invalid\_elems()),$
    $havoc\_partition(havoc\_subset(\texttt{knowledge} \cup generate\_invalid\_elems())), havoc\_number())$
function Add(e)
  assert $valid(e)$
  knowledge $\leftarrow$ knowledge $\cup \{e\}$
upon (BRB.Deliver(add(e))) **do**
  assert $valid(e)$
  knowledge $\leftarrow$ knowledge $\cup \{e\}$
function EpochInc(h)
  return
upon (BRB.Deliver(epinc(h))) **do**
  nothing
upon (SBC[h].SetDeliver($propset$)) **do**
  knowledge $\leftarrow$ knowledge $\cup \{e : e \in propset \wedge valid(e)\}$
upon (SBC[h].Inform($prop$)) **do**
  knowledge $\leftarrow$ knowledge $\cup \{e : e \in prop \wedge valid(e)\}$
function Start
  while true **do**
    BRB.Broadcast$(add(havoc_element(\texttt{knowledge} \cup generate_invalid_elems())))$
    ||
    BRB.Broadcast$(epinc(havoc_number()))$
    ||
    SBC$[havoc_number()]$.Propose$(havoc_subset(\texttt{knowledge} \cup generate_invalid_elems()))$
    ||
    nothing

- EpochInc(h) represents the invocation of function EpochInc(h),
- SBC[h].Propose($prop$) represents that the proposal $prop$ was made for the h instance of SBC,
- SBC[h].Inform($prop$) represents the reception of proposal $prop$,
- SBC[h].SetDeliver($propset$) represents that $propset$ is the result of the h instance of SBC,
- SBC[h].Consensus($propset$) is the internal event that denotes that consensus for epoch h is reached, and that set $propset$ was decided for that epoch,
- nop represents an event where nothing happens.

All events, except Consensus and nop, happen in a particular server, and thus, they have an attribute `server` that returns the corresponding server where they were triggered.

Network. We model the network Δ as a map from servers to tuples of the form (`sent`, `pending`, `received`). For a server s, $\Delta(s)$.`sent` is the sequence of messages sent from server s to other servers, $\Delta(s)$.`pending` is a multiset that contains all messages sent to s that it has not processed yet, and $\Delta(s)$.`received` is the sequence of messages received and processed by server s. The state of the network is modified when servers send or receive messages. Functions send() and receive() model such changes in the network. When server s sends a message m using BRB.Broadcast(m), send() adds m to the `sent` sequence of s and to the `pending` multiset of all other servers. Similarly, when server s proposes set $prop$ in the h instance of SBC, send() adds message $m = (h, prop)$ to the `sent` sequence of s and to the `pending` multiset of all servers. When servers receive a message m, receive() removes m from their `pending` multiset and inserts m in their `received` sequence.

We denote with Γ the *model* that represents the execution of a Setchain maintained by n processes implementing Alg. Fast out of which $1 \leq f < n/3$ are Byzantine servers.

A *configuration* $\Phi = (\Sigma, \Delta, H, K)$ for model Γ consists of: a state Σ mapping correct process to their local state; a network Δ containing messages exchanged between processes; a partial map H from epoch numbers to sets of elements (the consented history reached so far), and a set of valid elements K that have been disclosed to a Byzantine process.

The initial configuration $\Phi_0 = (\Sigma_0, \Delta_0, H_0, K_0)$ is such that $\Sigma_0(s)$ is the initial state of every correct process s, Δ_0 is the empty network, H_0 is the empty map, and K_0 is the empty set.

An event *ev* is considered *enabled* in configuration (Σ, Δ, H, K) based on the following conditions:

- $\mathtt{get}()$ and nop are always enabled,
- $\mathtt{add}(e)$ is enabled if e is valid and either $s = \mathtt{add}(e).\mathtt{server}$ is a Byzantine server or $e \notin \Sigma(s).S$,
- BRB.Broadcast(x) is enabled if BRB.Broadcast$(x).\mathtt{server}$ is a Byzantine server and either $x = \mathtt{epinc}(h)$ or $x = \mathtt{add}(e)$ with $e \in K$ or e invalid,
- BRB.Deliver$(\mathtt{add}(e))$ is enabled if $\mathtt{add}(e) \in \Delta(s).\mathtt{pending}$ and e is valid,
- EpochInc(h) is enabled if either $s =$ EpochInc$(h).\mathtt{server}$ is a Byzantine server or $h = \Sigma(s).\mathtt{epoch} + 1$,
- BRB.Deliver$(\mathtt{epinc}(h))$ is enabled if $\mathtt{epinc}(h) \in \Delta(s).\mathtt{pending}$ and either $s =$ BRB.Deliver$(\mathtt{epinc}(h)).\mathtt{server}$ is a Byzantine server, $h < \Sigma(s).\mathtt{epoch} + 1$, or $h = \Sigma(s).\mathtt{epoch} + 1$ plus s has not proposed anything for the h instance of SBC (i.e., no message of the form $(h, prop)$ is in $\Delta(s).\mathtt{sent}$),
- SBC$[h]$.Propose$(prop)$ is enabled if SBC$[h]$.Propose$(prop).\mathtt{server}$ is a Byzantine server and all valid elements in *prop* are known by Byzantine servers: $\{e \in prop : valid(e)\} \subseteq K$,
- SBC$[h]$.Inform$(prop)$ is enabled if $(h, prop) \in \Delta(s).\mathtt{pending}$,
- SBC$[h]$.SetDeliver$(propset)$ is enabled if $H(h) = propset$ and either $s = ev.\mathtt{server}$ is a Byzantine server or $h = \Sigma(s).\mathtt{epoch} + 1$,
- SBC$[h]$.Consensus$(propset)$ is enabled if $H(h-1)$ is defined, $H(h)$ is undefined, at least one process proposed a set before ($\exists s, h, prop : (h, prop) \in \Delta(s).\mathtt{sent}$), and *propset* is a subset of the union of all elements proposed for the h instance of SBC ($propset \subseteq \bigcup_r \{p : (h, p) \in \Delta(r).\mathtt{sent}\}$).

The *effect* of an enabled event *ev* on a configuration (Σ, Δ, H, K) results in the configuration $(\Sigma', \Delta', H', K')$. The updates for each component are as follows:

- For the set K', if $ev.\mathtt{server}$ is a Byzantine server, then $K' = K \cup valid\_elements(ev)$; otherwise $K' = K$. Where function $valid\_elements(ev)$ calculates the set of valid elements related to the event *ev*.
- For network Δ', updates depend on the type of event *ev*:
  - If event *ev* is $\mathtt{add}(e)$ and $s = ev.\mathtt{server}$ is a correct server, then $\Delta' = \mathsf{send}(\Delta, \mathtt{add}(e), s)$;
  - if event *ev* is BRB.Deliver$(\mathtt{add}(e))$, then $\Delta' = \mathsf{receive}(\Delta, \mathtt{add}(e), ev.\mathtt{server})$;
  - If event *ev* is EpochInc(h) and $s = ev.\mathtt{server}$ is a correct server, then $\Delta' = \mathsf{send}(\Delta, \mathtt{epinc}(h), s)$;
  - If event *ev* is BRB.Deliver$(\mathtt{epinc}(h))$ and $s = ev.\mathtt{server}$ is a Byzantine server or $h < \Sigma(s).\mathtt{epoch} + 1$, then $\Delta' = \mathsf{receive}(\Delta, \mathtt{epinc}(h), s)$
  - If event *ev* is BRB.Deliver$(\mathtt{epinc}(h))$ and $s = ev.\mathtt{server}$ is a correct server and $h = \Sigma(s).\mathtt{epoch} + 1$ then let *ps* be the set of elements in s without an epoch, $ps = \Sigma(s).S \setminus \bigcup_k^{h-1} \Sigma(s).H(k)$, in the new network $\Delta' = \mathsf{send}(\mathsf{receive}(\Delta, \mathtt{epinc}(h), s), (h, ps), s)$

- If event ev is BRB.Broadcast(m), then $\Delta' = \mathsf{send}(\Delta, m, ev.\mathtt{server})$;
- If event ev is SBC$[h]$.Propose$(prop)$, then $\Delta' = \mathsf{send}(\Delta, (h, prop), ev.\mathtt{server})$;
- If event ev is SBC$[h]$.Inform$(prop)$, then $\Delta' = \mathsf{receive}(\Delta, (h, prop), ev.\mathtt{server})$;
- otherwise $\Delta' = \Delta$.

- For the state map Σ', updates depend on whether the event $ev.\mathtt{server}$ is Byzantine or correct:
  - If event $ev.\mathtt{server}$ is Byzantine then $\Sigma' = \Sigma$.
  - If event $s = ev.\mathtt{server}$ is a correct server, the state is updated according to the type of event:
    * ev = BRB.Deliver(add(e)), then $\Sigma' = \Sigma \oplus \{s \mapsto (\Sigma(s).S \cup \{x : x = e \wedge \mathrm{valid}(x)\}, \Sigma(s).H, \Sigma(s).\mathrm{epoch})\}$.
    * ev = SBC$[h]$.SetDeliver$(propset)$ then $\Sigma' = \Sigma \oplus \{s \mapsto (\Sigma(s).S \cup E, \Sigma(s).H \cup \{\langle h, E\rangle\}, h)\}$ with $E = \{e : e \in propset, \mathrm{valid}(e) \wedge e \notin \Sigma(s).H\}$
    * otherwise $\Sigma' = \Sigma$.
- For the epoch map H':
  - If the event is SBC$[h]$.Consensus$(propset)$, then $H'(h) = propset$ and $H'(x) = H(x)$ for $x \neq h$.
  - otherwise $H' = H$.

If event ev is enabled at configuration (Σ, Δ, H, K) and $(\Sigma', \Delta', H', K')$ is the resulting configuration after applying the effect of ev to (Σ, Δ, H, K), then we write $(\Sigma, \Delta, H, K) \xrightarrow{ev} (\Sigma', \Delta', H', K')$.

Definition 1 (Valid Trace in Γ). *A valid trace in model* Γ *is an infinite sequence* $(\Sigma_0\Delta_0, H_0, K_0) \xrightarrow{ev_0} (\Sigma_1, \Delta_1, H_1, K_1) \xrightarrow{ev_1} \ldots$ *such that* $(\Sigma_0, \Delta_0, H_0, K_0)$ *is the initial configuration.*

We denote with Γ' the *model* that represents the execution of a Setchain that is maintained by $n - f$ correct servers implementing Alg. Fast and one server b implementing Alg. 8. A configuration in model Γ' is a tuple $(\Sigma, \Delta, H, \mathcal{T})$ where $\mathcal{T}$ is the local state of server b, and Σ, Δ and H are defined as in model Γ. The local state of server b consists in storing the knowledge harnessed by all Byzantine processes in model Γ.

The initial configuration $\Phi'_0 = (\Sigma_0, \Delta_0, H_0, \mathcal{T}_0)$ is such that $\Sigma_0(s)$ is the initial state of every correct process, $\mathcal{T}_0$ is the empty set, Δ_0 is the empty network, and H_0 is the empty map.

A valid configuration in Γ' follows the same principles as in model Γ. For correct processes, we have the same rules as in the previous model. The rules for process b are similar to the ones for Byzantines processes in the previous model. The difference is that here when process b consumes an event, all valid elements contained in the event are stored in b's local state $\mathcal{T}$, so it can use valid elements to produce events non-deterministically.

The effect of an event in Γ' is defined as follows. Given a configuration $(\Sigma, \Delta, H, \mathcal{T})$ where event ev is enabled in $(\Sigma, \Delta, H, \mathcal{T})$, the effect of ev is a configuration $(\Sigma', \Delta', H', \mathcal{T}')$ such that:

- For Δ', Σ' and H' the effect is as in Γ.
- For $\mathcal{T}'$:
  - If event $ev.\mathtt{server} = b$ then $\mathcal{T}' = \mathcal{T} \cup valid\_elements(ev)$
  - otherwise $\mathcal{T} = \mathcal{T}'$.

Note that the only "Byzantine" process now only annotates all elements that it discovers. The definition of valid trace is analogous as for Γ.

Definition 2 (Valid Trace in Γ'). *A valid trace in model* Γ' *is an infinite sequence* $(\Sigma_0, \Delta_0, H_0, \mathcal{T}_0) \xrightarrow{ev_0} (\Sigma_1, \Delta_1, H_1, \mathcal{T}_1) \xrightarrow{ev_1} \ldots$ *such that* $(\Sigma_0, \Delta_0, H_0, \mathcal{T}_0)$ *is the initial configuration.*

The main difference between Def 1 and Def. 2 is that now we as capturing what a "Byzantine" process could do (Alg. 8) and we update its state accordingly.

We aim to show that models Γ and Γ' are observational equivalent, meaning that external users cannot distinguish both models under the assumption that Byzantine processes may share information.

In order to prove this equivalence, we show that for each valid trace in one model, there is a valid trace in the other model such that corresponding configurations are indistinguishable. Two configurations $\Phi = (\Sigma, \Delta, H, K)$ and $\Phi' = (\Sigma', \Delta', H', \mathcal{T}')$ are observational equivalent, denoted $\Phi \sim \Phi'$, if and only if (1) every correct process has the same local state in both configurations, (2) the network are observational equivalents [5], (3) $\mathcal{T}$ and K contain the same elements, and (4) the histories reached by consensus are the same in both configurations.

The main idea is that since Byzantine processes share their knowledge outside the network, we can replace them by one non-deterministic process (b in model Γ') capable of taking every possible action that Byzantine processes can take. Hence, our definition of observational equivalent are based on mapping every Byzantine action in model Γ to possible an action of process b in model Γ'. Additionally, we map every single b action in model Γ' to a sequence of actions for the Byzantine processes in model Γ (one for each Byzantine process). To that end, we define the *stuttering extension* of a trace σ as the trace σ^{st} which adds $f - 1$ events nop after each event in σ.

Theorem 8.1. *For every valid trace in model Γ there exists a valid trace in model Γ' such that the corresponding configurations are indistinguishable.*

Proof. The proof consists on, given a valid trace σ in model Γ, construct a valid trace σ' in Γ'[6]. The construction of σ' is done by induction, ensuring that at each step the corresponding configurations are observational equivalent. In symbols, $\sigma_i \sim \sigma'_i$ for all $i \geq 0$. The initial configurations in both models are already equivalent.

The inductive step is done through a case analysis in the events that happen in σ. The inductive hypothesis assumes that for $n \geq 0$, σ' is defined up to n and $\sigma_i \sim \sigma'_i$ for $0 \leq i < n$. Consider the event ev_n such that $\sigma_n \xrightarrow{ev_n} \sigma_{n+1}$. We will show that exists an event ev'_n and a configuration σ'_{n+1} such that $\sigma'_n \xrightarrow{ev'_n} \sigma'_{n+1}$ and $\sigma'_{n+1} \sim \sigma_{n+1}$.

If ev_n happens in a correct server s, then $ev'_n = ev_n$ is also enabled in σ'_n, since whether an event that happens in a correct server is enabled or not in a given configuration depends only on the local state of s, the network related to s and the history of consensus reached, and all these components are the same in σ_n and σ'_n. Since both models run the same algorithm, events that happen in correct servers preserve the observational equivalence between configurations (see [4, Lemma 9]). Thus, the new configurations are also observational equivalent.

If ev_n happens in a Byzantine process then we consider the same event except that ev'_n happens at process b. Depending on configurationσ'_n , two cases arises: either ev'_n is enabled in σ'_n or not. In the former case, the effect of these events is to extend the Byzantine knowledge in each configuration with the valid elements discovered in the event, and apply the same combination of functions receive and send to each network. Thus, the new configurations are also observational equivalent (see [4, Lemma 10]). In the latter case, it must be the case that ev_n represents the reception of message that is pending in server ev_n.`server` in configuration σ_n but not in server b in configuration σ'_n. This means that b already consumed that message and knows about its valid elements. Thus,

[5]Intuitively, two networks are observational equivalent when they are the same for correct processes, and if a Byzantine process consumes (or sends) an event as long as process b also consumes (or sends) the same event. A formal definition can be found in [4].

[6]A more detailed proof can be found in [4].

configuration σ_{n+1} is also observational equivalent to configuration in σ'_n (see [4, Lemma 11]). Then, we define ev'_n as the nop event that is enabled in all configurations and does not have any effect.

If ev_n is a Consensus event, then $ev'_n = ev_n$ is enabled in configuration σ'_n since the history of consensus reached is the same as the one for σ_n and the observational equivalence between networks guarantee that if some server made a proposal in one configuration another server made the same proposal in the other configuration. The new configurations in both model extend the history of consensus reached in the same way, therefore they remain observational equivalent.

Finally, if the event is nop, then it is also enabled in configuration σ'_n, and it does not have any effect in both model. Thus, the new configurations remain observational equivalent. □

We prove now the other direction.

Theorem 8.2. *For every valid trace σ' in model Γ' there exists a valid trace σ in model Γ such that each configuration in σ is indistinguishable with corresponding state in the stuttering extension of σ' .*

Proof. The proof is by induction on the valid trace σ' and follows a reasoning similar to the one in the previous theorem. The main difference is that here for each event in σ' we have to find f events in model Γ, as we are considering the stuttering extension of σ'. We again proceed by case analysis in the events in σ'. There are two cases:

(1) If the event represents the reception of a message in process b, then we consider f events. Each event represents the reception of the message in one of the Byzantines processes in model Γ. Since the networks are observationally equivalent, events pending in process b are also pending in all Byzantine processes, therefore the events are also enabled in the configuration in model Γ. The effect of each event is to move messages from the `pending` multiset to the `received` list of the process where it happens and to annotate new valid elements discovered in the process. Thus, the corresponding configurations are observational equivalent.

(2) Otherwise, we consider the same event in model Γ followed by $f - 1$ nop events. With an analysis analogous to the one in the previous theorem, it can be shown that the considered events are enabled in the corresponding configurations, and that the resulting configurations are observational equivalent in both models. □

The main result of this section is that properties for model Γ hold if and only if they also hold in model Γ'. That is, one can prove properties for traces in model Γ' where all components are well-defined with only one Byzantine process and the result directly translate to traces in model Γ with multiple Byzantine processes with instantaneous communication. Our modeling and these results opens the door to apply mechanized formal verification to prove correctness properties of the Setchain algorithms.

9 CONCLUDING REMARKS AND FUTURE WORK

We presented in this paper 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 three distributed implementations, where the most efficient one uses Byzantine Reliable Broadcast and RedBelly Set Byzantine Consensus. Our empirical evaluation suggests that the performance of inserting elements in Setchain is three orders of magnitude faster than consensus. Also, we proved that the behavior of the Byzantine server nodes can be modeled by a collection of simple interactions with BRB and SBC and we introduced a non-deterministic process that encompasses these interactions. This modeling paves the way to use formal reasoning that is not equipped for Byzantine reasoning to reason about Setchain.

Future work includes developing the motivating applications listed in the introduction, including mempool logs using Setchains and L2 faster optimistic rollups. Setchain can also be used to alleviate front-running attacks. The mempool stores the transactions requested by users, so observing the mempool allows us to predict future operations. *Front-running* is the action of observing transaction request and maliciously inject transactions to be executed before the observed ones [10, 31] (by paying a higher fee to miners). 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 encryption scheme, and participant decrypting servers decrypt requests after they are chosen for execution by miners once the order has already been fixed.

We will also study how to equip blockchains with Setchain (synchronizing blocks and epochs) to allow smart-contracts to access the Setchain as part of their storage.

An important remaining problem is how to design a payment system for clients to pay for the usage of Setchain (even if a much smaller fee than for the blockchain itself). Our Setchain exploits a specific partial orders that relaxes the total order imposed by blockchains. As future work, we will explore other partial orders and their uses, for example, federations of Setchain, and one Setchain per smart-contract.

ACKNOWLEDGMENTS

We would like to than the anonymous reviewers for their useful comments. This work was funded in part by PRODIGY Project (TED2021-132464B-I00)—funded by MCIN/AEI/10.13039/501100011033/ and the European Union NextGenerationEU/PRTR—by DECO Project (PID2022-138072OB-I00)—funded by MCIN/AEI/10.13039/501100011033 and by the ESF+—and by a research grant from Nomadic Labs and the Tezos Foundation.

REFERENCES

[1] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. 2014. Zerocash: Decentralized Anonymous Payments from Bitcoin. In *Proc. of S&P'14*. 459–474. https://doi.org/10.1109/SP.2014.36

[2] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. 2014. Succinct Non-Interactive Zero Knowledge for a von Neumann Architecture. In *Proc. of USENIX Sec.'14*. USENIX, 781–796. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/ben-sasson

[3] Gabriel Bracha. 1987. Asynchronous Byzantine Agreement Protocols. *Inf. Comput.* 75, 2 (1987), 130–143. https://doi.org/10.1016/0890-5401(87)90054-X

[4] Margarita Capretto, Martín Ceresa, Antonio Fernández Anta, Antonio Russo, and César Sánchez. 2022. Improving Blockchain Scalability with Setchain data-type. *arXiv* (2022).

[5] Tushar Deepak Chandra and Sam Toueg. 1996. Unreliable Failure Detectors for Reliable Distributed Systems. *J. ACM* 43, 2 (mar 1996), 225–267. https://doi.org/10.1145/226643.226647

[6] Vicent Cholvi, Antonio Fernández Anta, Chryssis Georgiou, Nicolas Nicolaou, Michel Raynal, and Antonio Russo. 2021. Byzantine-tolerant Distributed Grow-only Sets: Specification and Applications. In *Proc. of FAB'21*. 2:1–2:19.

[7] Tyler Crain, Christopher Natoli, and Vincent Gramoli. 2021. Red Belly: A Secure, Fair and Scalable Open Blockchain. In *Proc. of S&P'21*. 466–483. https://doi.org/10.1109/SP40001.2021.00087

[8] F. Cristian, H. Aghili, R. Strong, and D. Volev. 1995. Atomic Broadcast: from simple message diffusion to Byzantine agreement. In *25th Int'l Symp. on Fault-Tolerant Computing*. 158–179. https://doi.org/10.1109/FTCSH.1995.532668

[9] 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. 2016. On Scaling Decentralized Blockchains. In *Financial Crypto. and Data Security*. Springer, 106–125.

[10] Philip Daian, Steven Goldfeder, T. Kell, Yunqi Li, X. Zhao, Iddo Bentov, Lorenz Breidenbach, and A. Juels. 2020. Flash Boys 2.0: Frontrunning in Decentralized Exchanges, Miner Extractable Value, and Consensus Instability. *Proc. of S&P'20* (2020), 910–927.

[11] Hung Dang, Tien Tuan Anh Dinh, Dumitrel Loghin, Ee-Chien Chang, Qian Lin, and Beng Chin Ooi. 2019. Towards Scaling Blockchain Systems via Sharding. In *Proc. of SIGMOD'19*. ACM, 123—-140. https://doi.org/10.1145/3299869.3319889

[12] Xavier Défago, André Schiper, and Péter Urbán. 2004. Total Order Broadcast and Multicast Algorithms: Taxonomy and Survey. *ACM Comput. Surv.* 36, 4 (dec 2004), 372–421. https://doi.org/10.1145/1041680.1041682
[13] Alan A.A. Donovan and Brian W. Kernighan. 2015. *The Go Programming Language.* Adison-Wesley.
[14] Antonio Fernández Anta, Chryssis Georgiou, Maurice Herlihy, and Maria Potop-Butucaru. 2021. *Principles of Blockchain Systems.* Morgan & Claypool Publishers.
[15] Antonio Fernández Anta, Kishori Konwar, Chryssis Georgiou, and Nicolas Nicolaou. 2018. Formalizing and implementing distributed ledger objects. *ACM Sigact News* 49, 2 (2018), 58–76.
[16] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. 1985. Impossibility of Distributed Consensus with One Faulty Process. *JACM* 32, 2 (1985), 374–382. https://doi.org/10.1145/3149.214121
[17] Rachid Guerraoui, Petr Kuznetsov, Matteo Monti, Matej Pavlovic, and Dragos-Adrian Seredinschi. 2019. The Consensus Number of a Cryptocurrency. In *Proc. of PODC'19.* ACM, 307–316. https://doi.org/10.1145/3293611.3331589
[18] Maxim Jourenko, Kanta Kurazumi, Mario Larangeira, and Keisuke Tanaka. 2019. SoK: A Taxonomy for Layer-2 Scalability Related Protocols for Cryptocurrencies. Cryptology ePrint Archive, Paper 2019/352. https://eprint.iacr.org/2019/352.
[19] Harry Kalodner, Steven Goldfeder, Xiaoqi Chen, S. Matthew Weinberg, and Edward W. Felten. 2018. Arbitrum: Scalable, private smart contracts. In *27th USENIX Security Symposium.* USENIX Assoc., 1353–1370. https://www.usenix.org/conference/usenixsecurity18/presentation/kalodner
[20] Jae Kwon and Ethan Buchman. 2019. Cosmos whitepaper. https://cosmos.network/resources/whitepaper.
[21] Zamani Mahdi, Mahnush Movahedi, and Mariana Raykova. 2018. RapidChain: Scaling Blockchain via Full Sharding. In *Proc. of CSS'18* (Toronto, Canada). ACM, 931—-948. https://doi.org/10.1145/3243734.3243853
[22] Satoshi Nakamoto. 2009. Bitcoin: a peer-to-peer electronic cash system.
[23] Joseph Poon and Thaddeus Dryja. 2016. The Bitcoin Lightning Network: Scalable Off-Chain Instant Payments. https://lightning.network/lightning-network-paper.pdf.
[24] Michel Raynal. 2018. *Fault-Tolerant Message-Passing Distributed Systems: An Algorithmic Approach.* Springer Cham, Switzerland. https://doi.org/10.1007/978-3-319-94141-7
[25] Robinson, Dan and Konstantopoulos, Georgios. 2020. *Ethereum is a Dark Forest.* https://medium.com/@danrobinson/ethereum-is-a-dark-forest-ecc5f0505dff
[26] Muhammad Saad, Laurent Njilla, Charles Kamhoua, Joongheon Kim, DaeHun Nyang, and Aziz Mohaisen. 2019. Mempool optimization for Defending Against DDoS Attacks in PoW-based Blockchain Systems. In *Proc. of ICBC'19.* 285–292. https://doi.org/10.1109/BLOC.2019.8751476
[27] Muhammad Saad, My T. Thai, and Aziz Mohaisen. 2018. POSTER: Deterring DDoS Attacks on Blockchain-Based Cryptocurrencies through Mempool Optimization. In *Proc. of ASIACCS'18.* ACM, 809–-811. https://doi.org/10.1145/3196494.3201584
[28] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Convergent and Commutative Replicated Data Types. *Bulletin- European Association for Theoretical Computer Science* 104 (June 2011), 67–88. https://hal.inria.fr/hal-00932833
[29] Nick Szabo. 1996. Smart Contracts: Building Blocks for Digital Markets. *Extropy* 16 (1996).
[30] The ZeroMQ authors. 2021. ZeroMQ. https://zeromq.org https://zeromq.org.
[31] Christof Ferreira Torres, Ramiro Camino, and Radu State. 2021. Frontrunner Jones and the Raiders of the Dark Forest: An Empirical Study of Frontrunning on the Ethereum Blockchain. In *Proc of USENIX Sec.'21.* 1343–1359. https://www.usenix.org/conference/usenixsecurity21/presentation/torres
[32] Shobha Tyagi and Madhumita Kathuria. 2021. *Study on Blockchain Scalability Solutions.* ACM, 394–401. https://doi.org/10.1145/3474124.3474184
[33] Ke Wang and Hyong S. Kim. 2019. FastChain: Scaling Blockchain System with Informed Neighbor Selection. In *Proc. of IEEE Blockchain'19.* 376–383. https://doi.org/10.1109/Blockchain.2019.00058
[34] Gavin Wood. 2014. Ethereum: A secure decentralised generalised transaction ledger. Ethereum project yellow paper, 151.
[35] Gavin Wood. 2016. Polkadot: Vision for a heterogeneous multi-chain framework. White Paper.
[36] Cheng Xu, Ce Zhang, Jianliang Xu, and Jian Pei. 2021. SlimChain: Scaling Blockchain Transactions through off-Chain Storage and Parallel Processing. *Proc. VLDB Endow.* 14, 11 (jul 2021), 2314–2326. https://doi.org/10.14778/3476249.3476283