# Byzantine-tolerant Distributed Grow-only Sets: Specification and Applications

**Vicent Cholvi**
Universitat Jaume I, Spain

**Antonio Fernández Anta**
IMDEA Networks Institute, Spain

**Chryssis Georgiou**
Dept. of Computer Science, University of Cyprus, Cyprus

**Nicolas Nicolaou**
Algolysis Ltd., Cyprus

**Michel Raynal**
IRISA, France & PolyU, Hong Kong

**Antonio Russo**
IMDEA Networks Institute, Spain

## Abstract

In order to formalize Distributed Ledger Technologies and their interconnections, a recent line of research work has formulated the notion of Distributed Ledger Object (DLO), which is a concurrent object that maintains a totally ordered sequence of records, abstracting blockchains and distributed ledgers. Through DLO, the Atomic Appends problem, intended as the need of a primitive able to append multiple records to distinct ledgers in an atomic way, is studied as a basic interconnection problem among ledgers.

In this work, we propose the *Distributed Grow-only Set object* (DSO), which instead of maintaining a sequence of records, as in a DLO, maintains a set of records in an immutable way: only Add and Get operations are provided. This object is inspired by the Grow-only Set (G-Set) data type which is part of the Conflict-free Replicated Data Types. We formally specify the object and we provide a consensus-free Byzantine-tolerant implementation that guarantees eventual consistency. We then use our Byzantine-tolerant DSO (BDSO) implementation to provide consensus-free algorithmic solutions to the Atomic Appends and Atomic Adds (the analogous problem of atomic appends applied on G-Sets) problems, as well as to construct consensus-free Single-Writer BDLOs. We believe that the BDSO has applications beyond the above-mentioned problems.

**2012 ACM Subject Classification** Theory of computation → Design and analysis of algorithms

**Keywords and phrases** Grow-only Sets, Distributed Ledgers, Blockchains, Atomic appends

**Digital Object Identifier** 10.4230/OASIcs.FAB.2021.2

**Funding** Partially supported by French ANR project ByBLoS (ANR-20-CE25-0002-01), Government of Madrid (CM) grant EdgeData-CM (P2018/TCS4499, cofunded by FSE & FEDER), and Spanish Ministry of Science and Innovation grant ECID (PID2019-109805RB-I00, cofunded by FEDER).

## 1 Introduction

Blockchains (as termed by Nakamoto in [18]) or Distributed Ledger Technologies (DLTs) (as used in [10] and [20]) became one of the most trendy data structures following the introduction of crypto-currencies [18] and their recent application in finance and token-economy. Despite their early wide adoption, little was known initially about the fundamental construction and semantic properties of DLTs. A number of research groups attempted to provide rigorous definitions to characterise the fundamental properties of DLTs as those used in Bitcoin and beyond [1, 10, 11]. Among those, Fernández Anta et al. [10], was the first to identify and





provide a formal definition of a reliable concurrent object, termed *Distributed Ledger Object* (DLO), which conveys the essential building block for many DLTs. In particular, a DLO maintains a sequence of records, and supports two basic operations: append and get. The append operation is used to add a new record at the end of the sequence, while the get operation returns the whole sequence. Implementations of DLOs under client and server crashes were proposed in [10], and under Byzantine failures in [6].

The introduction to many different DLT systems have led multiple studies [6, 9, 14, 16] to investigate the possibility of DLT interoperability, i.e., the ability for an action to be applied over a set of DLTs, rather than in a single DLT at a time. Using the DLO formalism, [9] introduced the *Atomic Appends problem*, in which several clients have a "composite" record (a set of semantically-linked "basic" records) to append. Each basic record has to be appended to a different DLO, and it must be guaranteed that either all basic records are appended to their DLOs or none of them is appended.

Consider, for example, two clients $A$ and $B$, where $A$ buys a car from $B$. Record $r_A$ includes the transfer of the car's digital deed from $B$ to $A$, and $r_B$ includes the transfer from $A$ to $B$ of the agreed amount in some digital currency. $\text{DLO}_A$ is a ledger maintaining digital deeds and $\text{DLO}_B$ maintains transactions in some pre-agreed digital currency. So, while the two records are mutually dependent, they concern different DLOs. Hence, the Atomic Appends problem requires that *either* record $r_A$ is appended in $\text{DLO}_A$ and record $r_B$ is appended in $\text{DLO}_B$, *or* no record is appended in the corresponding DLOs.

In the work presented in [9], the authors assumed that clients may fail by crashing and showed that for some cases the existence of an intermediary is necessary. They materialized such an intermediary by implementing a specialized DLT, termed *Smart* DLO (SDLO). Using the SDLO, the authors solved the Atomic Appends problem in a client competitive asynchronous environment, in which any number of clients, and up to $f$ servers implementing the DLOs, may crash. A subsequent work solved the problem assuming Byzantine failures [6], by introducing the notion of *Byzantine Distributed Ledger Objects* (BDLO). Solutions for implementing BDLOs were presented, with each solution relying on an underlying Byzantine Total-order Broadcast Service (BToB) [7, 8, 17]. Using BToB and an intermediary SBDLO the authors demonstrated how Atomic Appends may be achieved in systems that suffer Byzantine failures. However, BToB is a strong primitive, and requires consensus to be solved. So one may ask: *Is it possible to implement Atomic Appends without solving consensus?*

It was shown in [13] that cryptocurrencies do not need consensus to be implemented. From a theoretical point of view, it was shown in [12] that, assuming one process per account, the consensus number of cryptocurrencies is 1. A non-sequential specification of money transfer was introduced in [2]. It follows that Byzantine transactional systems do not necessarily need consensus, but rather can be implemented on top of less powerful data structures. In a similar manner, in this work, we observe that intermediary S(B)DLOs and strong primitives like BToB [17], may not be necessary to allow interoperability between multiple DLOs. Note that the goal of the intermediate S(B)DLO is to collect the records to be appended atomically, so that when all the records involved are in the S(B)DLO, then the actual records are appended in their respective DLOs. It is apparent that, for *Atomic Appends*, the order of the records in the intermediary data structure is not important, but rather the membership property required redirects to a *set* data structure.

A relevant distributed set data structure was presented by Shapiro et al. in [21] with the introduction of Conflict-Free Replicated Data Types (CRDTs). A CRDT is a data structure that can be replicated in multiple network locations. CRDTs have the property that each replica can be updated independently and concurrently, but it is always mathematically possible to resolve any inconsistencies between any pair of replicas, leading eventually all

the replicas to a consistent converged value when the communication between the replica hosts is stabilized. A *Grow-Only Set* (G-Set) is such a CRDT, that supports operations add and lookup only. The add operation modifies the local state of the object by a union of the value of the set with the element we want to insert. Since add is based on union, and union is commutative, the G-Set implementation converges. In [21] (and other subsequent works), implementations of G-Sets where given in a crash-prone environment. In order to utilise a G-Set in more practical setups (like the ones in cryptocurrencies) we need to examine whether such data structure is possible when Byzantine failures are present in the system.

Chai and Zhao [5] have considered the implementation of CRDTs against Byzantine failures. In particular, they describe possible threats that clients and servers can either face or cause to CRDTs, and they show a possible solution to fulfil CRDT requirements in that failure model. Their solution relies on an external synchronization service for two main purposes: to guarantee linearizable *reads* and *writes*, and to prevent server partitions caused by Byzantine behaviour. As a consequence, multiple Byzantine failures or slow processes may lead their approach to essentially always run their "state synchronization" mechanism letting the whole data structure rely on the synchronisation service. For the implementation of the synchronisation service they either utilize a central entity, or solve consensus over a distributed set of nodes.

**Contributions.** In this work we examine whether G-Sets can be implemented when Byzantine processes are assumed in the system, without using consensus. We show that an implementation of an eventually consistent [22] G-Set is possible, and we demonstrate how such data structure can be used to solve Atomic Appends and other related problems. In particular, our itemized contributions are the following:

- Provide a formal definition of a Byzantine Grow-only Set Object (BDSO). [Section 2]
- Provide an implementation for an eventually consistent BDSO in an asynchronous message passing system[1]. We consider such a consistency model since, although it provides weaker guarantees than other consistency models, it is easier and more efficient to implement, while being powerful enough to be used in the type of applications we consider (described next). [Section 3]
- Use BDSOs to implement:
  - Consensus-free Byzantine Atomic Appends. [Section 4.1]
  - Consensus-free Byzantine *Atomic Adds*. This is the analogous problem of atomic appends where records must be added in an atomic way to different BDSOs. This problem could be applicable in blockchain-like systems in which the ordering of the records is not important; what is important is that the records are added in the corresponding unordered blockchains (G-Sets). An example could be a system of G-Sets that implement personal calendars, so the records in the sets are meetings. Then, fixing a two-person meeting would imply an Atomic Add of the meeting data in the calendar of both persons. [Section 4.2]
  - Consensus-free single-writer BDLOs. This data structure can be suitable to implement whatever system that requires total order among data produced by a single writer. A punch in/out system for a company is an example of such an application in which a single writer, the employee, appends records only to his/her own ledger of presences. A cryptocurrency can be another suitable application, with one BDLO per account, because of the need to order transactions in relation to money transfers issued by the only transaction signer. [Section 4.3]

[1] Note that in such a system deterministic consensus can't be solved.

# 2 The G-Set Object

In this section we provide the fundamental definition of a concurrent G-Set object.

## 2.1 Concurrent Objects and the G-Set Object

An *object type* $T$ specifies ($i$) the set of *values* (or states) that any object $O$ of type $T$ can take, and ($ii$) the set of *operations* that a process can use to modify or access the value of $O$. An object $O$ of type $T$ is a *concurrent object* if it is a shared object accessed by multiple processes [15, 19]. Each operation on an object $O$ consists of an *invocation* event and its unique matching *response* event, that must occur in this order. A *history* of operations on $O$, denoted by $H_O$, is the sequence of invocation and response events, starting with an invocation event. (The sequence order of a history reflects the real time ordering of the events.) We say that a history $H'_O$ *extends* a history $H_O$, if $H_O$ is a prefix of $H'_O$.

An operation $\pi$ is *complete* in a history $H_O$, if $H_O$ contains both the invocation and the matching response. A history $H_O$ is *complete* if it contains only complete operations; otherwise it is *partial* [15, 19]. An operation $\pi$ *precedes* an operation $\pi'$ (or $\pi'$ *succeeds* $\pi$), denoted by $\pi \rightarrow \pi'$, in $H_O$, if the response event of $\pi$ appears before the invocation event of $\pi'$ in $H_O$. Two operations are *concurrent* if none precedes the other. A complete history $H_O$ is *sequential* if it contains no concurrent operations, i.e., it is an alternative sequence of matching invocation and response events, starting with an invocation and ending with a response event. A partial history is sequential, if removing its last event (that must be an invocation) makes it a complete sequential history.

A *sequential specification* of an object $O$, describes the behavior of $O$ when accessed sequentially. In particular, the sequential specification of $O$ is the set of all possible sequential histories involving solely object $O$ [19].

A *G-Set* $\mathcal{GS}$ is a concurrent object that maintains a set $\mathcal{GS}.S$ of *records* and supports two operations (available to any process $p$): (i) $\mathcal{GS}.\mathsf{get}_p()$, and (ii) $\mathcal{GS}.\mathsf{add}_p(r)$. A *record* is any value drawn from an alphabet $A$. A process $p$ invokes a $\mathcal{GS}.\mathsf{get}_p()$ operation to obtain the set $\mathcal{GS}.S$ of records stored in the G-Set object $\mathcal{GS}$ [2], and $p$ invokes a $\mathcal{GS}.\mathsf{add}_p(r)$ operation to insert a new record $r$ in $\mathcal{GS}.S$. Initially, the set $\mathcal{GS}.S$ is empty. Deleting or changing a record from $\mathcal{GS}.S$ is not possible, as our objective is for the set to be immutable with respect to record modifications of any kind.

▶ **Definition 1.** *The* sequential specification *of a G-Set $\mathcal{GS}$ over the sequential history $H_{\mathcal{GS}}$ is defined as follows. Let the initial value of $\mathcal{GS}.S = \emptyset$. If at the invocation event of an operation $\pi$ in $H_{\mathcal{GS}}$ the value of the set $\mathcal{GS}.S = V$, then:*

1. *if $\pi$ is a $\mathcal{GS}.\mathsf{get}_p()$ operation, then the response event of $\pi$ returns $V$, and*
2. *if $\pi$ is a $\mathcal{GS}.\mathsf{add}_p(r)$ operation, then at the response event of $\pi$, the value of the set in G-Set $\mathcal{GS}$ is $\mathcal{GS}.S = V \cup \{r\}$.*

By comparing the sequential specification of a G-Set, as defined above, with the sequential specification of a Ledger Object as defined in [10, Definition 1] (also see Appendix A), it follows that a Ledger is an *ordered* G-Set.

[2] We define only one operation to access the value of the G-Set for simplicity. In practice, other operations will also be available, like *lookup(r)* to check if a record $r$ is in $\mathcal{GS}.S$.

## 2.2 Distributed G-Set Objects

We now define distributed G-Set objects, DSO for short, and the class of eventually consistent DSOs. These definitions are general and do not rely on the properties of the underlying distributed system, nor on the type of failures that may occur.

A **distributed G-Set object** (DSO) is a concurrent G-Set object that is implemented in a distributed manner. In particular, a DSO is *implemented* by a set of (possibly distinct and geographically dispersed) computing devices, that we refer as *servers*. Each server usually maintains a local copy (replica) of the DSO. We refer to the processes that invoke the get and add operations of the distributed G-Set as *clients*.

Distribution and replication intend to ensure availability and survivability of the G-Set, in case a subset of the servers fails (by crashing or acting maliciously). At the same time, they raise the challenge of maintaining *consistency* among the different views that different clients get of the DSO[3]. Consistency semantics need to be in place to precisely describe the allowed values that a get operation may return when it is executed concurrently with other get or add operations.

We now specify the properties of DSO with respect to *eventual consistency* [22]. These properties require that if an $\mathsf{add}(r)$ operation completes, then *eventually* all $\mathsf{get}()$ operations return sets that contain record $r$. In a similar way, other consistency guarantees such as sequential, session, causal and atomic consistencies could be formally defined.

▶ **Definition 2.** *A DSO $\mathcal{GS}$ is* **eventually consistent** *if, given any history $H_{\mathcal{GS}}$,*

*(a) EC-Safety: let $S$ be the set of records returned by any complete operation $\pi = \mathsf{get}() \in H_{\mathcal{GS}}$. For each $r \in S$, there is an operation $\mathsf{add}(r)$ whose invocation event appears before the response event of $\pi$ in $H_{\mathcal{GS}}$, and*

*(b) EC-Liveness: for every complete operation $\mathcal{GS}.\mathsf{add}(r) \in H_{\mathcal{GS}}$, there exists a history $H'_{\mathcal{GS}}$ that extends $H_{\mathcal{GS}}$ such that, for every history $H''_{\mathcal{GS}}$ that extends $H'_{\mathcal{GS}}$, every complete operation $\mathcal{GS}.\mathsf{get}()$ in $H''_{\mathcal{GS}} \setminus H'_{\mathcal{GS}}$ returns a set that contains $r$.*

At this point, we would like to remark that, although eventual consistency provides weaker consistency guarantees when compared, for example, with linearizability [15], it is easier and more efficient to implement, while it is powerful enough to be used in the type of applications that we later consider (see Section 4).

## 2.3 Distributed Setting and Byzantine-tolerant DSO

We consider a distributed setting consisting of processes (clients and servers) and an underlying communication graph in which each process can communicate with every other process.

**Asynchrony.** Both processing and communication are asynchronous. Therefore, each process proceeds at its own speed, which can vary arbitrarily and remains always unknown to the other processes. Message transfer delays are arbitrary but finite and remain always unknown to the processes.

**Failure Model.** Processes (clients and servers) can fail arbitrarily, i.e., they can be Byzantine. Specifically, we assume a *Byzantine system* in which the number of servers that can arbitrarily fail is bounded by $f$, and in which the total number of servers, $n$, is at least $3f + 1$. For clients we assume that any of them can be Byzantine. We assume reliable channels between non-Byzantine (correct) processes. Specifically, no message is lost, duplicated or modified.

[3] This tradeoff is actually captured by the well-known CAP Theorem [4].

**Public and private keys.** We assume that each process $p$ (client or server) has a pair of public and private keys, and that the public keys have been distributed reliably to all the processes that may interact with each other. Hence, we discard the possibility of spurious or fake processes (there cannot be Sybil attacks). We also assume that messages sent by any process (server or client) are authenticated, so that messages corrupted or fabricated by Byzantine processes are detected and discarded by correct processes [8]. Communication channels between correct processes are reliable but asynchronous.

**Byzantine-tolerant DSOs.** Our first aim is to propose an algorithm that implement an eventual-consistent DSO $\mathcal{GS}$ in a Byzantine asynchronous system. Here we present the properties that a DSO should satisfy with respect to *correct processes*, given that Byzantine processes may return any arbitrary set or add any arbitrary record:

- *Byzantine Completeness (BC)*: All the get() and add() operations invoked by correct clients eventually complete.
- *Byzantine Eventual Consistency (BEC)*: This is the property of Definition 2 with respect to all operations invoked by correct clients and the add($r$) operations that insert the records $r$ returned by get() operations invoked by correct clients.

In the remainder, we say that a DSO is *Byzantine Tolerant*, denoted BDSO, and eventually consistent if it satisfies properties BC and BEC.

**Byzantine Reliable Broadcast.** The algorithms presented in the next section to implement BDSOs are based on an underlying Byzantine Reliable Broadcast (BRB) service [3, 20], which ensures that a message sent by a correct process is received by all correct processes, and that all correct processes eventually receive the same set of messages. The service provides two operations, BRB-broadcast and BRB-delivery; the first broadcasts a message to all processes, and the second delivers a message that was previously broadcast. The service is used by the servers, and from their point of view, the BRB service guarantees the following properties (as given in [20]):

- *Validity*: if a correct process $p_i$ BRB-delivers a message $m$ from a correct process $p_j$, then $p_j$ BRB-broadcast $m$.
- *Integrity*: a message is BRB-delivered at most once by a correct server.
- *Termination 1 (local)*: if a correct process BRB-broadcasts a message, it BRB-delivers it.
- *Termination 2 (global)*: if a correct process BRB-delivers a message, all correct processes BRB-deliver it.

Validity relates outputs to inputs. Validity and integrity concern safety. Termination is on the fact that messages must be BRB-delivered; it concerns liveness. It follows (cf. [20]) that all correct processes BRB-deliver the same set of messages, which includes all the messages they BRB-broadcast.

## 3 Eventually Consistent BDSO Implementation

In this section we provide the implementation of eventually consistent distributed G-Sets in an asynchronous distributed system with Byzantine failures. The implementation builds on a generic deterministic Byzantine-tolerant reliable broadcast service [3, 20], which provides the properties given in the previous section. Our implementation is *optimally resilient*, in the sense that it can tolerate up to $f$ Byzantine servers, out of $n \geq 3f + 1$ servers.

Algorithm 1 presents the code of a client process, while Algorithm 2 presents the code of a server. We now present a high level description of how the two algorithms together implement an eventually consistent BDSO.

**Algorithm 1** Client API and algorithm for Eventually Consistent Byzantine-tolerant Distributed G-Set Object $\mathcal{GS}$. Code for Client $p$.

```
1:  Init: c ← 0
2:  function GS.get( )                                          ▷ Invocation event
3:      c ← c + 1
4:      send request GET(c, p) to 3f + 1 different servers
5:      wait responses GETRESP(c, i, S_i) from 2f + 1 different servers
6:      S ← {r : record r is in at least f + 1 sets S_i}
7:      return S                                                ▷ Response event
8:  function GS.add(r)                                          ▷ Invocation event
9:      c ← c + 1
10:     send request ADD(c, p, r) to 2f + 1 different servers
11:     wait responses ADDRESP(c, i, ACK) from f + 1 different servers
12:     return ACK                                              ▷ Response event
```

**Algorithm 2** Server algorithm for Eventually Consistent Byzantine-tolerant Distributed G-Set Object. Code for Server $i$.

```
1:  Init: S_i ← ∅
2:  receive (GET(c, p)) from process p                          ▷ Signature of p is validated
3:      send response GETRESP(c, i, S_i) to p
4:  receive (ADD(c, p, r)) from process p                       ▷ Signature of p is validated
5:      if (r ∉ S_i) then
6:          BRB-broadcast(PROPAGATE(i, ADD(c, p, r)))
7:          wait until r ∈ S_i
8:      send response ADDRESP(c, i, ACK) to p
9:  upon (BRB-deliver(PROPAGATE(j, ADD(c, p, r)))) do           ▷ Signatures of j and p are validated
10:     if (r ∉ S_i) and (ADD(c, p, r) was received from f + 1 different servers j) then
11:         S_i ← S_i ∪ {r}
```

- When processing a $\mathcal{GS}$.add($r$) operation a client sends ADD messages to a set of $2f + 1$ servers, which guarantees that at least $f + 1$ correct servers process it. These correct servers broadcast the record $r$ to all servers using the BRB service, which leads to all correct servers $i$ adding $r$ to their replicas $S_i$ of the set. When $f + 1$ acknowledgement messages are received from the servers, the operation completes.
- When processing a $\mathcal{GS}$.get() operation, a client need to ensure that the elements he returns have been received from at least 1 correct server. For this reason the client returns an element only if it was present in responses from $f+1$ different server. In order to avoid the malicious behavior of $f$ colluding servers that never return a correctly added element, at least $2f + 1$ responses are needed out of which take the $f + 1$ consistent GETRESP containing the element. So, since $2f + 1$ are required, at least $3f + 1$ GET messages must be sent in order to always eventually get the number of needed responses.
- Every server $i$ maintains a replica $S_i$ of the set $\mathcal{GS}.S$. When server $i$ receives a GET($c$, $p$) message from a process $p$ it returns its current set $S_i$ to $p$. When $i$ receives a message ADD($c$, $p$, $r$) from $p$, it makes sure $r$ has been included in its replica $S_i$ before sending an acknowledgment. Server $i$ adds a record $r$ to its replica $S_i$ only if a corresponding add request has been processed by at least one correct server. This is guaranteed by the BRB service and the requirement of receiving PROPAGATE($j$, ADD($c$, $p$, $r$)) from $f + 1$ different servers. This also prevents Byzantine servers from adding spurious records in the set of correct servers. The properties of the BRB service also guarantee that once a record $r$ is

delivered, then all correct servers will eventually add record $r$ to their replicas.

We now provide the complete proof that the combination of Algorithms 1 and 2 implement an eventually consistent BDSO. In the proofs we consider that an operation $\pi$ is invoked in Lines 2 or 8 of Algorithm 1, and responds in Lines 7 or 12 (resp.) of the same algorithm. Let us first show that Byzantine Completeness holds, i.e., that all operations invoked by correct processes eventually complete.

▶ **Lemma 3.** *Algorithms 1 and 2 guarantee Byzantine Completeness (BC) in a system in which at most $f$ out of $n \geq 3f + 1$ servers are Byzantine.*

**Proof.** Consider an operation $\mathcal{GS}.\mathsf{get}_p()$ invoked by a correct client $p$. We claim that the operation eventually completes. From Algorithm 1, Line 4, $p$ sends a request GET($c$, $p$) to $3f + 1$ servers and waits for responses GETRESP($c$, $i$, $S_i$) from $2f + 1$ different servers. From the $3f + 1$ servers to which the request is sent, at most $f$ can be Byzantine, so at least $2f + 1$ are correct servers that will eventually receive the GET($c$, $p$) message. These servers will immediately send the corresponding response GETRESP($c$, $i$, $S_i$) to $p$ (Line 3 of Algorithm 2). When these responses are received eventually, the waiting in Line 5 of Algorithm 1 will end. Since there is no other waiting condition, the operation will execute the return instruction and complete.

Consider now an operation $\pi = \mathcal{GS}.\mathsf{add}_p(r)$ invoked by a correct client $p$. Then, the request ADD($c$, $p$, $r$) is sent to $2f + 1$ servers (Algorithm 1, Line 10), and $p$ waits until responses ADDRESP($c$, $i$, ACK) are received from $f + 1$ different servers. Since at most $f$ servers can be Byzantine, at least $f + 1$ correct servers will receive and process the request. We prove that all these correct servers will send the corresponding response, the waiting in Line 11 will end, and operation $\pi$ will complete.

Let us consider the set $C$ of correct servers that receive request ADD($c$, $p$, $r$). Assume first that there is some server $i \in C$ that has $r \in S_i$ when the request is received and processed. Then, server $i$ sends immediately response ADDRESP($c$, $i$, ACK) to $p$. Moreover, $r$ was inserted in $S_i$ in Line 11 of Algorithm 2, which implies that $i$ received via BRB-deliver at least $f + 1$ messages PROPAGATE() from different servers containing ADD($c$, $p$, $r$) requests. From the *Termination 2* property of the BRB service, all correct processes will receive the same $f + 1$ messages PROPAGATE(). Consider any other correct server $j \in C$ that receives request ADD($c$, $p$, $r$). If $r \in S_j$ when the request is received and processed, server $j$ sends the response ADDRESP($c$, $j$, ACK) to $p$ immediately. Otherwise, $r \notin S_j$ when the request is received and processed, and $j$ waits in Line 7. From the above argument, eventually $r$ will be inserted in $S_j$, the waiting will end, and $j$ will send response ADDRESP($c$, $j$, ACK) to $p$.

Assume now that no correct server $i \in C$ has $r \in S_i$ when it receives request ADD($c$, $p$, $r$). Then, all the (at least $f + 1$) correct servers in $C$ that receive and process the request invoke BRB-broadcast(PROPAGATE($i$, ADD($c$, $p$, $r$))) and start waiting in Line 7. From the *Termination 1* property of the BRB-service, if a correct server BRB-broadcasts a message, it also eventually BRB-delivers it. Moreover, from *Termination 2*, if it BRB-delivers the message, all correct servers also BRB-deliver it. So each correct server $i \in C$ will process in Lines 9-11 messages PROPAGATE($j$, ADD($c$, $p$, $r$)) from at least $f + 1$ different servers $j$. Hence, server $i$ will insert $r$ in $S_i$ in Line 11, the waiting will end, and $i$ will send response ADDRESP($c$, $i$, ACK) to $p$. ◀

▶ **Theorem 4.** *Algorithms 1 and 2 implement an Eventually Consistent* BDSO*, in a system in which at most $f$ out of $n \geq 3f + 1$ servers are Byzantine.*

**Proof.** We need to prove that Algorithms 1 and 2 guarantee Byzantine Completeness (BC) and Byzantine Eventual Consistency (BEC). BC is shown to be satisfied in Lemma 3.

Regarding Byzantine Eventual Consistency, we need to demonstrate properties (a) and (b) of Definition 2 with respect to all the operations invoked by correct clients and the add($r$) operations that insert the records $r$ returned in the get() operations invoked by correct clients. Let $H_{\mathcal{GS}}$ be any history including only invocation and response events of these operations.

*Property (a):* Consider a complete operation $\pi = \mathsf{get}_p() \in H_{\mathcal{GS}}$ invoked by a correct client $p$, let $S$ be the set returned by $\pi$, and consider any $r \in S$. From Line 6 of Algorithm 1, $r$ belongs to at least $f+1$ sets $S_i$ received in responses GETRESP($c$, $i$, $S_i$) from a set $C$ of different servers. All these responses must have been sent before the response event of $\pi$ (Line 7 of Algorithm 1).

Observe that $C$ contains at least one correct server $i$. This mean that some correct server $i$ had $r \in S_i$ when it sent the response GETRESP($c$, $i$, $S_i$). A server $i$ only adds a record to its local set $S_i$ if that record was BRB-delivered in PROPAGATE($j$, ADD($c'$, $p'$, $r$))) from $f+1$ different servers $j$ (Line 10 of Algorithm 2). From the Validity property of the BRB service, this means that at least $f+1$ servers called BRB-broadcast(PROPAGATE($j$, ADD($c'$, $p'$, $r$)) in Line 6. Again, since at least one of them is correct, at least one invocation of BRB-broadcast was done by a process because it previously received a request ADD($c'$, $p'$, $r$) from client $p'$. Hence the invocation of add($r$) must have preceded the reception of this request, and by transitivity must have preceded the response event of $\pi$.

*Property (b):* This property holds if, for every complete operation $\mathcal{GS}.\mathsf{add}(r) \in H_{\mathcal{GS}}$, there exists a time $t$ after which every $\mathcal{GS}.\mathsf{get}()$ operation invoked after $t$ returns sets $S$ that contains $r$. Let us first consider a complete operation $\pi = \mathcal{GS}.\mathsf{add}_p(r) \in H_{\mathcal{GS}}$ invoked by a client $p$ (which can be correct or Byzantine). We claim that there is some correct server $i$ that eventually adds record $r$ to its replica $S_i$. This is true when $p$ is Byzantine, since that is the requirement for an add($r$) operation of a Byzantine client to be considered.

On the other hand, if $p$ is correct, let us assume for contradiction that no correct server $i$ adds record $r$ to its replica $S_i$. Process $p$ sends request ADD($c$, $p$, $r$) to $2f+1$ servers, out which at least $f+1$ are correct. By assumption, $r \notin S_j$ when each of these servers $j$ processes the request, and hence all of them execute BRB-broadcast(PROPAGATE($j$, ADD($c$, $p$, $r$))) (Line 6 of Algorithm 2). Then, from the *Termination 1* and *Termination 2* properties of the BRB service, some correct server $i$ will BRB-deliver at least $f+1$ messages PROPAGATE($j$, ADD($c$, $p$, $r$)) from different servers $j$, and then record $r$ will be added to $S_i$ in Line 11. This is a contradiction, and some correct server $i$ eventually adds record $r$ to its replica $S_i$ when client $p$ is correct.

Hence, we have that, independently of whether $p$ is correct, some correct server $i$ added record $r$ to its set $S_i$. Observe that a correct process $i$ only adds records to its replica $S_i$, in Line 11, when BRB-deliver at least $f+1$ messages PROPAGATE($j$, ADD($c$, $p$, $r$)) from different servers $j$. Then, if $i$ adds $r$ to $S_i$, from the *Termination 2* property all correct servers will eventually BRB-deliver at least $f+1$ messages PROPAGATE($j$, ADD($c$, $p$, $r$)) from different servers $j$, and they will all add $r$ to their replicas.

Let $t$ be the first time all correct servers have $r$ in their corresponding replica. Then, for every $\mathcal{GS}.\mathsf{get}()$ operation invoked after $t$, the responses from correct servers collected in Line 5 of Algorithm 1 have replicas $S_i$ with record $r$. Since there at least $f+1$ responses from correct servers, in Line 6 $r$ is included in the set $S$, which is then returned by $\mathcal{GS}.\mathsf{get}()$. ◀

## 4 Applications of BDSOs

In this section we demonstrate the usability of BDSOs by using them to provide consensus-free solutions to the Atomic Appends and Atomic Adds problems, as well as a consensus-free construction of a Single-Writer Byzantine-tolerant Distributed Ledger Object (BDLO).

## 4.1 The Atomic Appends Problem

The *Atomic Appends* problem was introduced in [10] as a basic interconnection problem among distributed ledgers (DLOs); see Appendix A for basic definitions with respect to DLOs. Informally, Atomic Appends requires that several records must be appended in their corresponding DLOs, so that either *all* records are appended (each in the appropriate DLO) or *none* is appended to any DLO. In [6], the problem was formulated (and solved) in the presence of Byzantine servers and clients.

**Definition of the problem.** For completeness, we provide the formal definition as given in [6]. A record $r$ *depends* on a record $r'$ if $r$ may be appended on its intended BDLO, say $\mathcal{L}$, only if $r'$ is appended on its intended BDLO, say $\mathcal{L}'$. Two records, $r$ and $r'$ are *mutually dependent* if $r$ depends on $r'$ and $r'$ depends on $r$.

▶ **Definition 5** (2-AtomicAppends [6]). *Consider two clients, $p$ and $q$, with mutually dependent records $r_p$ and $r_q$. We say that records $r_p$ and $r_q$ are* appended atomically *in BDLO $\mathcal{L}_p$ and BDLO $\mathcal{L}_q$, respectively, when:*

- *AA-safety (AAS):* The record $r_p$ of a correct client $p$ is appended in $\mathcal{L}_p$ only if the record of the other client $q$ (which may be correct or not) is also appended in $\mathcal{L}_q$.
- *AA-liveness (AAL):* If both $p$ and $q$ are correct, then both records are appended eventually.

Observe that it is not possible to prevent a faulty client $q$ from appending its record $r_q$, even if the correct client $p$ does not append its record. What the safety property AAS guarantees is that the opposite cannot happen. This is analogous of the property in atomic cross-chain swaps [14] that a correct process cannot end up worse than at the beginning.

We say that an algorithm *solves* the 2-AtomicAppends problem[4] under a given system, if it guarantees properties AAS and AAL of Definition 5 in every execution. Since we consider Byzantine failures, our system model with respect to the Atomic Appends problem is such that the correct processes want to proceed with the append of the records (to guarantee liveness AAL), while the Byzantine processes may try to get correct clients to append without the Byzantine clients doing so (to prevent safety AAS).

**Prior solution.** The solution of 2-AtomicAppends in [6], following the work in [10], uses an auxiliary, special purpose BDLO, called Smart BDLO (SBDLO) to aggregate and coordinate the append of multiple records. In a nutshell, the solution in [6] is as follows. Consider two clients, $p$ and $q$, that wish to append atomically two mutually dependent records, $r_p$ and $r_q$, in BDLOs $\mathcal{L}_p$ and $\mathcal{L}_q$, respectively. Then, they both send matching *atomic append requests* to the SBDLO. Once both requests are received by the SBDLO (otherwise the atomic append never takes place), the servers implementing the SBDLO proceed to append each record to the appropriate BDLOs. In particular, the servers of the SBDLO now become clients issuing the corresponding appends to the servers implementing the DBLOs $\mathcal{L}_p$ and $\mathcal{L}_q$ (each BDLO could be implemented by different servers, as these are essentially different distributed ledger systems). The whole process involves several algorithms: the algorithm run by the clients to issue the atomic append request, the algorithm run by servers to implement the SBDLO, and the algorithm run by the servers of the SBDLO (as clients) with the servers of each individual BDLO. Once both append operations are completed, the SBDLO servers acknowledge this to clients $p$ and $q$. It is shown that the combination of these algorithms guarantee Properties AAS and AAL above, despite having Byzantine servers and clients.

[4] The *k-AtomicAppends* problem, for $k \geq 2$, is a generalization of the 2-AtomicAppends that can be defined in the natural way: $k$ clients, with $k$ mutually dependent records, to be appended to $k$ BDLOs. To keep the presentation simple, we focus in the case of $k = 2$.

**Algorithm 3** API for the 2-AtomicAppend of records $r_p$ and $r_q$ in ledgers $\mathcal{L}_p$ and $\mathcal{L}_q$ by clients $p$ and $q$, respectively, using SBDSO $\mathcal{GS}$. Code for Client $p$.

```
1: function AtomicAppends(p, {p, q}, r_p, L_p, r_q)
2:     GS.add(⟨p, {p, q}, r_p, L_p, r_q⟩)
3:     return ACK
4: // Client p will know the Atomic Appends operation was completed successfully when it receives
   notifications from f + 1 different SBDSO servers. //
```

**Algorithm 4** Server algorithm for Smart Byzantine-tolerant DSO. Code for Server $i$.

```
1: Init: S_i ← ∅
2: receive (GET(c, p)) from process p                          ▷ Signature of p is validated
3:     send response GETRESP(c, i, S_i) to p
4: receive (ADD(c, p, r)) from process p                       ▷ Signature of p is validated
5:     if (r ∉ S_i) then
6:         BRB-broadcast(PROPAGATE(i, ADD(c, p, r)))
7:         wait until r ∈ S_i
8:     send response ADDRESP(c, i, ACK) to p
9: upon (BRB-deliver(PROPAGATE(j, ADD(c, p, r)))) do     ▷ Signatures of j and p are validated
10:    if (r ∉ S_i) and (ADD(c, p, r) was received from f + 1 different servers j) then
11:        S_i ← S_i ∪ {r}
12:        if (r.v = ⟨p, {p, q}, r_p, L_p, r_q⟩) and
13:         (∃r' ∈ S_i : r'.v = ⟨q, {p, q}, r_q, L_q, r_p⟩) then
14:            L_p.append(r_p); L_q.append(r_q)
15:            Notify clients p and q that records r_p and r_q have been appended to L_p and L_q
```

**Our approach.** In this work we treat the part of the individual BDLOs ($\mathcal{L}_p$ and $\mathcal{L}_q$) implementations as black boxes and we focus on the auxiliary entity that is used for coordinating the atomic append requests. In [6], the SBDLO, being a Distributed Ledger object, required the use of a Byzantine Total-order Broadcast [17] service. It was shown in [10] that consensus is required for implementing a (B)DLO; this is because of the strong prefix property of (B)DLOs (see Appendix A), which requires that records must be totally ordered. Hence, atomic appends was solved using consensus to implement the SBDLO. However, one can notice that in the auxiliary entity, the atomic append requests do not need to be totally ordered. It is sufficient to only keep track whether both requests have been made. In other words, *why keeping these requests in a sequence, and not in a set?*

In this respect, we show that instead of using a special purpose BDLO as the auxiliary entity, we can simply use a special purpose eventually consistent BDSO, which we will be referring as SBDSO. As we have seen in Section 3, eventually consistent BDSOs can be implemented without consensus (instead of a Byzantine total-order broadcast service, we use only a Byzantine reliable broadcast service), yielding a *consensus-free solution to Atomic Appends* (with respect to the actual atomic append requests).

**Our solution.** Algorithm 3 specifies how processes $p$ and $q$ delegate the task of appending their records in the respective ledgers. They do so by adding in the SBDSO a description of the Atomic Appends operation to be completed. Client $p$ uses the $\mathcal{GS}$.add operation to provide the SBDSO with the data it requires to complete the Atomic Appends, namely the participants in the Atomic Appends, the record $r_p$, the BDLO $\mathcal{L}_p$, and the record $r_q$ the other client is appending. (The other client must do the same.)

For the SBDSO, it suffices to implement an eventually consistent BDSO in which up to $f$ servers out of $n \geq 3f + 1$ are Byzantine, but that *only allows the creator of a record to add*

*it* (signatures are used for this purpose). Algorithm 4 describes the processing of the ADD message by the SBDSO. As expected, it is very similar to the implementation of a BDSO, but with an important difference: every time a record $r$ is added to the sequence $S_i$, it is checked whether a matching record $r'$ is already there. This is the case if $r.v = \langle p, \{p, q\}, r_p, \mathcal{L}_p, r_q \rangle$, and $r'.v = \langle q, \{p, q\}, r_q, \mathcal{L}_q, r_p \rangle$. If so, the corresponding append operations are issued in the respective BDLOs $\mathcal{L}_p$ and $\mathcal{L}_q$ (the implementation of this part is the one described in [6]). So, essentially the servers implementing the SBDSO, become proxies of clients $p$ and $q$, and once the above condition is met, they issue the corresponding appends. When these appends are successful, the servers implementing the ledgers $\mathcal{L}_p$ and $\mathcal{L}_q$, acknowledge the SBDSO servers. In turn, the SBDSO servers notify clients $p$ and $q$ that records $r_p$ and $r_q$ have been appended to $\mathcal{L}_p$ and $\mathcal{L}_q$, respectively. Clients $p$ and $q$ will know that the Atomic Appends operations was completed successfully when they receive these notifications from at least $f + 1$ different SBDSO servers.

▶ **Theorem 6.** *The combination of Algorithms 3 and 4 solves the* 2*-AtomicAppends problem.*

The proof follows from the one in [6], taking into consideration the above discussion.

***Remark:*** Following the approach described in [6, Section IV-B], the SBDSO can be replaced by a "classical" BDSO $\mathcal{GS}$ and the use of a set of "helper" processes. The helper processes take upon themselves the task of consulting $\mathcal{GS}$ periodically in order to find new matching descriptions of and Atomic Appends operation. When such a match is found, they complete the corresponding appends (as done in Lines 13-15 of Algorithm 4).

## 4.2 The Atomic Adds Problem

Inspired by the Atomic Appends problem, one could define the analogous problem on BDSOs, *Atomic Adds*: several records must be added in their corresponding BDSOs, and either all records are added (each in the appropriate BDSO) or none is added. The formal definition follows that of the Atomic Appends.

▶ **Definition 7** (2-AtomicAdds)**.** *Consider two clients, $p$ and $q$, with mutually dependent records*[5] *$r_p$ and $r_q$. We say that records $r_p$ and $r_q$ are* added atomically *in BDSO $\mathcal{GS}_p$ and BDSO $\mathcal{GS}_q$, respectively, when:*

- *AAd-safety (AAdS):* The record $r_p$ of a correct client $p$ is added in $\mathcal{GS}_p$ only if the record of the other client $q$ (which may be correct or not) is also added in $\mathcal{GS}_q$.
- *AAd-liveness (AAdL):* If both $p$ and $q$ are correct, then both records are added eventually.

The $k$-AtomicAdds problem can be defined in the natural way: $k$ clients, with $k$ mutually dependent records, to be appended to $k$ BDSOs. It is not difficult to see that a consensus-free algorithmic solution for this problem can be derived by simple modifications of our solution to the Atomic Appends problem and the use of the BDSO implementation of Section 3.

**Atomic Adds API and server code.** The Atomic Adds API, shown in Algorithm 5, is very close to Algorithm 3. The main difference is the content of the data to be added (since now we have G-Sets and not ledgers). The code run by the servers of SBDSO is the same as in Algorithm 4, with the difference that Lines 12 and 13 check for matching atomic add requests, and once found, in Line 14 will call the corresponding add operations, $\mathcal{GS}.\mathsf{add}(r_p)$ and $\mathcal{GS}.\mathsf{add}(r_q)$, which are implemented by the algorithms in Section 3. Note

[5] The definition of mutually dependent records is as in the case of Atomic Appends, but for BDSOs instead of BDLOs.

**Algorithm 5** API for the 2-AtomicAdds of records $r_p$ and $r_q$ in BDSOs $\mathcal{GS}_p$ and $\mathcal{GS}_q$ by clients $p$ and $q$, respectively, using SBDSO $\mathcal{GS}$. Algorithm for Client $p$.

```
1: function AtomicAdds(p, {p, q}, r_p, GS_p, r_q)
2:     GS.add(⟨p, {p, q}, r_p, GS_p, r_q⟩)
3:     return ACK
4: // Client p will know the Atomic Adds operation was completed successfully when it receives
   notifications from f + 1 different SBDSO servers. //
```

that the condition in Line 10 of Algorithm 2 may have to be expanded in order to prevent the (up to $f$) Byzantine servers that implement the SBDSO from adding spurious records in $\mathcal{GS}_p$ and $\mathcal{GS}_q$. This may be achieved adding a record $r$ in these DSOs only if at least $f + 1$ clients (the servers of the SBDSO) request it to be added, similarly as done in [6].

The sequence of events is now as described in the Atomic Appends solution, with the difference that no BDLOs are now involved, only BDSOs. Putting everything together, we obtain the following, whose proof details are omitted (it is essentially a restatement of the corresponding observations in the atomic appends proof in [6], and the correctness of the algorithms in Section 3):

▶ **Theorem 8.** *The combination of the API of Algorithm 1, the API of Algorithm 5, and the revised versions of Algorithms 2 and 4, yields a solution to the 2-AtomicAdds problem.*

As noted above, the SBDSO could be replaced by a "classical" BDSO and the use of a set of "helper" processes. See [6, Section IV-B] for this approach.

## 4.3 Consensus-free Single-Writer BDLO

The BDSO can also be used to implement a Single-Writer BDLO without relying on consensus. This is obtained with a BDSO that allows only a single writer process $w$ to add records, in which each record has an index determining its position in the BDLO sequence, and that does not allow adding more than one record with the same index. Allowing only add operations from $w$ is trivially achieved by validating the signature when a request is received by a server, and will not be done explicitly in our algorithms. To prove correctness we need to show that any execution of the Single-Writer BDLO $\mathcal{L}$ we implement satisfies the Byzantine Completeness and Byzantine Eventual Consistency properties, but redefined for the $\mathcal{L}$.append() and $\mathcal{L}$.get() operations, and sequences instead of sets (see Appendix A). Additionally, the Byzantine Strong Prefix property, as defined in [6], must also be satisfied.

▶ **Definition 9** (Byzantine Strong Prefix [6]). *If two correct clients of a BDLO $\mathcal{L}$ issue two $\mathcal{L}$.get() operations that return record sequences $S$ and $S'$ respectively, then either $S$ is a prefix of $S'$ or vice-versa.*

Algorithm 6 presents the API and the code executed by a client of the Single-Writer BDLO $\mathcal{L}$, while Algorithm 7 presents the code executed by the servers that implement it. These algorithms require that the number of servers $n$ satisfies $n \geq 4f + 1$. As can be seen, the append operation assigns an index $k$ to every record data $d$ appended by $w$, so the record added is in fact the pair $r = (k, d)$. Observe that Algorithms 6 and 7 are very similar to Algorithms 1 and 2, but have a few differences. (1) In Algorithm 6, $\mathcal{L}$.append($d$) adds an index $k$ to each record and sends the append requests to a potentially much larger set of $\lfloor n/2 \rfloor + 2f + 1$ servers, while $\mathcal{L}$.get() filters the set to be returned so it is a sequence of records with consecutive indices. (2) Algorithm 7 avoids appending different records with the same index $r.k$ by using this field for comparisons, keeping track in $T$ of the indices that

**Algorithm 6** Client API and algorithms for Eventually Consistent Single-Writer BDLO $\mathcal{L}$ with $n \geq 4f + 1$ and writer process $w$. Code for Client $p$.

```
1: Init: c ← 0, k ← 0
2: function L.get( )
3:     c ← c + 1
4:     send request GET(c, p) to 3f + 1 different servers
5:     wait responses GETRESP(c, i, S_i) from 2f + 1 different servers
6:     A ← {r : record r is in at least f + 1 sets S_i}
7:     S ← {r ∈ A : (r.k = 1) ∨ (∃r' ∈ A : r.k = r'.k + 1)}
8:     return sequence ⟨ρ_1, ..., ρ_m⟩, where m = |S| and r_ℓ = (ℓ, ρ_ℓ) ∈ S
9: function L.append(ρ)                          ▷ Can only be called by process w
10:    c ← c + 1, k ← k + 1
11:    r ← (k, ρ)
12:    send request ADD(c, w, r) to ⌊n/2⌋ + 2f + 1 different servers
13:    wait responses ADDRESP(c, i, ACK) from f + 1 different servers
14:    return ACK
```

**Algorithm 7** Server algorithm for Eventually Consistent Single-Writer BDLO $\mathcal{L}$ with $n \geq 4f + 1$ and writer process $w$. Code for Server $i$, and Writer $w$

```
1: Init: S_i ← ∅, T ← ∅
2: receive (GET(c, p)) from process p
3:     send response GETRESP(c, i, S_i) to p
4: receive (ADD(c, w, r)) from process w
5:     if (r.k ∉ T) then
6:         BRB-broadcast(PROPAGATE(i, ADD(c, w, r)))
7:         T ← T ∪ {r.k}
8:         wait until r ∈ S_i
9:     send response ADDRESP(c, i, ACK) to w
10: end receive
11: upon (BRB-deliver(PROPAGATE(j, ADD(c, w, r)))) do
12:     if (ADD(c, w, r) was received from ⌊n/2⌋ + f + 1 different servers j) then
13:         S_i ← S_i ∪ {r}
```

have been BRB broadcast, and collecting at least $\lfloor n/2 \rfloor + f + 1$ messages PROPAGATE($j$, ADD($c$, $w$, $r$)) before adding $r$ to the set. Observe that the requirement on $n$ comes from the fact that the append requests are sent to $\lfloor n/2 \rfloor + 2f + 1$ servers, and hence, $f < n/4$.

▶ **Theorem 10.** *Algorithms 6 and 7 implement an eventually consistent Single-Writer BDLO $\mathcal{L}$.*

**Proof.** We will first show Byzantine Completeness, then Byzantine Eventual Consistency and lastly Byzantine Strong Prefix.

***Byzantine Completeness***: Let us consider an $\mathcal{L}$.get() operation invoked by a correct client $p$. Then request GET($c$, $p$) is sent to $3f + 1$ different servers so at least $2f + 1$ correct ones will eventually send back their responses; in fact correct servers simply answer back in Line 3 of Algorithm 7 with a GETRESP($c$, $i$, $S_i$) containing their local $S_i$. Then, the condition of the wait operation in Line 5 is eventually satisfied and the operation completes.

Let us now assume that $w$ is correct, and consider an $\mathcal{L}$.append() operation. Then, requests ADD($c$, $w$, $r$) will be sent (Line 12 of Algorithm 6) to $\lfloor n/2 \rfloor + 2f + 1$ servers, so at least $\lfloor n/2 \rfloor + f + 1$ correct ones will receive it. Since $w$ is correct, it increments $k$ before sending

the ADD($c$, $w$, $r$) messages (Line 10 of Algorithm 6), so the same index $k$ is not used twice. Then, every correct process that receives ADD($c$, $w$, $r$) finds that $r.k \notin T$ (since $T$ is updated in Line 7 of Algorithm 7 only after this check). Hence, the BRB-broadcast(PROPAGATE($i$, ADD($c$, $w$, $r$))) in Line 6 is called at least by $\lfloor n/2 \rfloor + f + 1$ correct servers. For this reason, by the Termination properties of the BRB service, the condition in Line 12 will eventually be satisfied exactly once and record $r$ is inserted in the local set $S_i$ (Line 13 of Algorithm 7). So the condition in Line 8 of Algorithm 7 turns true and the response is sent back to the correct client $w$. Since this holds for at least $\lfloor n/2 \rfloor + f + 1$ correct servers that received the request, and $\lfloor n/2 \rfloor + f + 1 > f + 1$, the condition in Line 13 of Algorithm 6 will be satisfied and the append operation will terminate.

***Byzantine Eventual Consistency***: In order to demonstrate Byzantine Eventual Consistency we need to demonstrate Properties (a) and (b) of Definition 2 with respect to histories $H_{\mathcal{L}}$ that contain only events of get operations by correct clients and append operations of records that are returned in those get operations. Note that $\mathcal{L}$.append($\rho$) and $\mathcal{L}$.get() are considered in place of $\mathcal{GS}$.add($r$) and $\mathcal{GS}$.get().

- *Property (a):* Let $\mathcal{L}$.get be a complete operation in $H_{\mathcal{L}}$. Let $S$ be the set from where the sequence returned by $\mathcal{L}$.get is extracted. Then, from Line 7 of Algorithm 6, $\forall r \in S$ the client verified that $r$ belongs to $f + 1$ different sets $S_i$ (Line 6 of Algorithm 6) returned in a GETRESP($c$, $i$, $S_i$) by different servers. This means that at least a correct server has $r \in S_i$. A server only adds data to its local set $S_i$ if that data was BRB-delivered in PROPAGATE(-, ADD(-, -, $r$)) messages from $\lfloor n/2 \rfloor + f + 1$ different servers. Thanks to the Validity property of the BRB service, this means that at least $\lfloor n/2 \rfloor + f + 1$ servers called BRB-broadcast with that message. Again, at least $\lfloor n/2 \rfloor + 1$ of them are correct, and they called BRB-broadcast because they received ADD($c$, $p$, $r$) from client $w$. So, $\forall r = (k, \rho) \in S$, an $\mathcal{L}$.append($\rho$) invocation precedes the $\mathcal{L}$.get response.
- *Property (b):* This is equivalent to say that $\forall \rho$ such that $\mathcal{L}$.append($\rho$) $\in H_{\mathcal{L}}$, eventually there exist a time $t$ such that $\rho$ will be included in all the sequences returned by complete $\mathcal{L}$.get $\in H_{\mathcal{L}}$ invoked after $t$.

  Assume $w$ is Byzantine and consider an operation $\mathcal{L}$.append($\rho$) $\in H_{\mathcal{L}}$. Then, some $\mathcal{L}$.get() operation by a correct client returned a sequence with $r = (k, \rho)$, which means that it received at least $f + 1$ messages GETRESP($c$, $i$, $S_i$) in which $r \in S_i$. This means that at least one correct server $i$ had $r \in S_i$. Then, server $i$ BRB-delivered at least $\lfloor n/2 \rfloor + f + 1$ PROPAGATE(-, ADD(-, -, $r$)) messages, and by the Termination properties of the BRB service all correct servers $j$ will do as well, and will include $r$ in their local sets $S_j$. Then, any other get operation will always have $f + 1$ responses including $r$ from correct servers.

  Assume now that $w$ is correct. Then, it sends requests ADD($c$, $w$, $r$) with $r = (k, \rho)$ to at least $\lfloor n/2 \rfloor + 2f + 1$ servers, so that at least $\lfloor n/2 \rfloor + f + 1$ correct ones will process it calling BRB-broadcast in Line 6 of Algorithm 7. From the Termination properties of the BRB service, $\lfloor n/2 \rfloor + f + 1$ PROPAGATE(-, ADD(-, -, $r$)) messages coming from different servers will be eventually BRB-delivered to all correct servers. Then, all correct servers will eventually add $r$ to their local $S_i$ because of the fulfilment of $\lfloor n/2 \rfloor + f + 1$ requirement in Line 12 of Algorithm 7. $\mathcal{L}$.get(), on its side, returns $r$ if it was seen at least in $f + 1$ **out of** $2f + 1$ different responses. Since at most $f$ can have Byzantine behaviour and eventually all server will include $r$ in their local $S_i$, there will exist a moment in which $\mathcal{L}$.get() will always have $f + 1$ responses including $r$ from correct servers.

  We have shown that, independently of whether $w$ is correct, if $\rho$ is returned in some get operation of a correct client, eventually a record $r = (k, \rho)$ will be in all the sets $S_j$ of all correct servers $j$. Then, there exist a moment in which $r$ is definitely always part of temporary set $A$ in Line 6 of Algorithm 6 in all get operations. Now, in order to ensure

that $r$ is part of $S$, and the sequence returned, we need to demonstrate that Line 7 of client Algorithm 6 does not filter it, eventually. We proceed by induction. If $r.k = 1$ then record $r$ is included in $S$. If $r.k > 1$, assume the claim true for record $r' = (k-1, \rho')$. I.e., there is a time $t'$ after which $r'$ is always in $A$. Then, there is a time $t \geq t'$ in which both $r$ and $r'$ are always in $A$. After $t$ record $r$ will always be included in $S$ and returned by all get operations.

***Byzantine Strong Prefix***: Let $S = (r_0, ..., r_a)$ and $S' = (r'_0, ..., r'_b)$ the two sets from which the sequences returned by the two $\mathcal{L}$.get() operations are extracted in Line 8. Just as a convenience in notation, we will refer $r = (k, \rho)$ as $r_k = \rho$. Line 7 of the client Algorithm 6 ensures that records in $S$ and $S'$ can be ordered and that there are not missing element in the sequence. If $S$ and/or $S'$ are empty then one is trivially prefix of the other. So let's assume they both have at least one element and, without loss of generality, that $a \leq b$. Also, let us assume by way a contradiction that the sequence extracted from $S$ is not a prefix of the sequence from $S'$. This is equivalent to state that $\exists i \leq a : r_i \neq r'_i$. From Line 6 of Algorithm 6 we know that $r_j$ and $r'_j$ with $1 \leq j \leq a$ were returned at least by one correct server in their respective get operations. So, assuming that such an index $i$ exists means that at least two correct servers executed Line 13 of Algorithm 7 for the two records, respectively. This implies that, for both, the condition of Line 12 was true because they received messages PROPAGATE($-$, ADD($c$,$p$,$r_i$)) from a set $C$ of at least $\lfloor n/2 \rfloor + f + 1$ servers, and messages PROPAGATE($-$, ADD($c$,$p$,$r'_i$)) from a set $C'$ of at least $\lfloor n/2 \rfloor + f + 1$ servers. Note that each $C$ and $C'$ contains at least $\lfloor n/2 \rfloor + 1$ correct servers. It is obvious that broadcasters of these PROPAGATE messages must intersect in at least one correct server $j$. So, from the Validity property of the BRB service, at least correct server $j$ called both BRB-broadcast(PROPAGATE($j$, ADD($c$,$p$,$r_i$))) and BRB-broadcast(PROPAGATE($j$, ADD($c$,$p$,$r'_i$))). Line 5 of Algorithm 7 filters the received ADD($c$,$p$,$r$) request, so only if $r.k \notin T$ they are propagated via the BRB-broadcast. If so, Line 7 of Algorithm 7 adds $r.k$ to $T$ right after the BRB-broadcast. Assume, w.l.o.g., that $j$ received ADD($c$,$p$,$r_i$) before receiving ADD($c$,$p$,$r'_i$). As soon as $j$ BRB-broadcast PROPAGATE($i$, ADD($c$,$p$,$r_i$))), it added $r_i.k$ to $T$. Then, when it received ADD($c$,$p$,$r'_i$) it found that $r'_i.k \in T$, and BRB-broadcast(PROPAGATE($i$, ADD($c$,$p$,$r'_i$))) was not executed. But this contradicts our assumption that $\exists i \leq a : r_i \neq r'_i$. Hence, the sequence extracted from $S$ must be a prefix of the sequence from $S'$. ◀

## 5 Conclusions and Future Work

In this paper we formally define the notion of a Byzantine-tolerant Distributed G-Set Object (BDSO) and provide client and server algorithms to implement a consensus-free eventually consistent BDSO. Then we proceed with some use cases for BDSO. Building on the work in [6] and using BDSOs we provide a consensus-free solution to the Atomic Appends problem. Similarly, we provide a consensus-free solution to the Atomic Adds problem, the analogous problem that uses sets instead of ledgers. Finally, we show how a few modifications to the client and server algorithms of BDSO, enable to realise an eventual consistent Single-Writer Byzantine Distributed Ledger without solving consensus among servers but still guaranteeing the Byzantine Strong Prefix property. Single-Writer consensus-free BDLO can be suitable for many use cases, like implementing a cryptocurrency or a punch in/out system for employees of a company. These are scenarios where realising transactional systems in a Byzantine failure model through consensus may not provide reasonable performance, since the need of updating the system global status prevents sustaining a high throughput of operations. Our future plans include implementing and experimentally evaluating the algorithms proposed in this work, as well as specifying a cryptocurrency based on single-writer BDLOs.

## References


1. Anceaume, E., Pozzo, A. D., Ludinard, R., Potop-Butucaru, M., and Tucci Piergiovanni, S. Blockchain abstract data type. In *31st ACM on Symposium on Parallelism in Algorithms and Architectures, SPAA 2019, Phoenix, AZ, USA, June 22-24, 2019* (2019), ACM, pp. 349–358.
2. Auvolat, A., Frey, D., Raynal, M., and Taïani, F. Money transfer made simple: a specification, a generic algorithm, and its proof. *Bull. EATCS 132* (2020), 23–43.
3. Bracha, G. Asynchronous byzantine agreement protocols. *Inf. Comput. 75*, 2 (1987), 130–143.
4. Brewer, E. Cap twelve years later: How the "rules" have changed. *Computer 45*, 2 (2012), 23–29.
5. Chai, H., and Zhao, W. Byzantine fault tolerance for services with commutative operations. In *IEEE International Conference on Services Computing, SCC 2014, Anchorage, AK, USA, June 27 - July 2, 2014* (2014), IEEE Computer Society, pp. 219–226.
6. Cholvi, V., Fernandez Anta, A., Georgiou, C., Nicolaou, N., and Raynal, M. Atomic appends in asynchronous byzantine distributed ledgers. In *2020 16th European Dependable Computing Conference (EDCC)* (2020), pp. 77–84.
7. Coelho, P., Junior, T. C., Bessani, A., Dotti, F., and Pedone, F. Byzantine fault-tolerant atomic multicast. In *DSN 2018* (2018), IEEE, pp. 39–50.
8. Cristian, F., Aghili, H., Strong, R., and Dolev, D. Atomic broadcast: From simple message diffusion to byzantine agreement. *Information and Computation 118*, 1 (1995), 158 – 179.
9. Fernández Anta, A., Georgiou, C., and Nicolaou, N. Atomic appends: Selling cars and coordinating armies with multiple distributed ledgers. In *International Conference on Blockchain Economics, Security and Protocols, Tokenomics 2019, Paris, France* (2019), pp. 39–50.
10. Fernández Anta, A., Konwar, K. M., Georgiou, C., and Nicolaou, N. C. Formalizing and implementing distributed ledger objects. *SIGACT News 49*, 2 (2018), 58–76.
11. Garay, J. A., Kiayias, A., and Leonardos, N. The bitcoin backbone protocol: Analysis and applications. In *34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, EUROCRYPT 2015, Sofia, Bulgaria, April 26-30, 2015, Part II* (2015), pp. 281–310.
12. Guerraoui, R., Kuznetsov, P., Monti, M., Pavlovic, M., and Seredinschi, D. The consensus number of a cryptocurrency. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, PODC 2019, Toronto, ON, Canada, July 29 - August 2, 2019* (2019), ACM, pp. 307–316.
13. Gupta, S. *A non-consensus based decentralized financial transaction processing model with support for efficient auditing.* Arizona State University, 2016.
14. Herlihy, M. Atomic cross-chain swaps. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing, PODC 2018, Egham, United Kingdom, July 23-27, 2018* (2018), pp. 245–254.
15. Herlihy, M. P., and Wing, J. M. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS) 12*, 3 (1990), 463–492.
16. Koens, T., and Poll, E. Assessing interoperability solutions for distributed ledgers. *Pervasive and Mobile Computing 59* (2019), 101079.
17. Milosevic, Z., Hutle, M., and Schiper, A. On the reduction of atomic broadcast to consensus with byzantine faults. In *SRDS 2011* (2011), pp. 235–244.
18. Nakamoto, S. Bitcoin: A peer-to-peer electronic cash system. `https://bitcoin.org/bitcoin.pdf`, 2008. [Online; accessed 22-February-2021].
19. Raynal, M. *Concurrent Programming: Algorithms, Principles, and Foundations.* Springer, 2013.
20. Raynal, M. *Fault-Tolerant Message-Passing Distributed Systems - An Algorithmic Approach.* Springer, 2018.

21 Shapiro, M., Preguiça, N., Baquero, C., and Zawirski, M. Conflict-free replicated data types. In *13th International Symposium Stabilization, Safety, and Security of Distributed Systems, SSS 2011, Grenoble, France* (2011), Springer, pp. 386–400.
22 Vogels, W. Eventually consistent. *Commun. ACM 52*, 1 (2009), 40–44.

# Appendix

## A DLO Definitions

For the reader's convenience, we provide the basic definitions regarding Distributed Ledger Objects [10].

A ***ledger*** $\mathcal{L}$ is a concurrent object that stores a totally ordered sequence $\mathcal{L}.S$ of records and supports two operations (available to any process $p$): (i) $\mathcal{L}.\mathsf{get}_p()$, and (ii) $\mathcal{L}.\mathsf{append}_p(r)$. The sequential specification of a ledger $\mathcal{L}$ is as follows:

▶ **Definition 11.** *The* sequential specification *of a ledger $\mathcal{L}$ over the sequential history $H_{\mathcal{L}}$ is defined as follows. The value of the sequence $\mathcal{L}.S$ of the ledger is initially the empty sequence. If at the invocation event of an operation $\pi$ in $H_{\mathcal{L}}$ the value of the sequence in ledger $\mathcal{L}$ is $\mathcal{L}.S = V$, then:*

1. *if $\pi$ is an $\mathcal{L}.\mathsf{get}_p()$ operation, then the response event of $\pi$ returns $V$, and*
2. *if $\pi$ is an $\mathcal{L}.\mathsf{append}_p(r)$ operation, then at the response event of $\pi$, the value of the sequence in ledger $\mathcal{L}$ is $\mathcal{L}.S = V \| r$ (where $\|$ is the concatenation operator).*

A ***Distributed Ledger Object***, DLO for short, is a concurrent ledger object that is implemented in a distributed manner. In particular, the ledger object is implemented by *servers*, and *clients* invoke the $\mathsf{get}()$ and $\mathsf{append}()$ operations.

▶ **Definition 12.** *A DLO $\mathcal{L}$ is* **eventually consistent** *if, given any history $H_{\mathcal{L}}$,*

*(a) Let $S$ be the sequence of records returned by any complete operation $\pi = \mathsf{get}() \in H_{\mathcal{L}}$ and $\rho_i$ the generic record that belongs to $S$. For each $\rho_i \in S$ then $H_{\mathcal{L}}$ contains $\mathsf{append}(\rho_j)$ for $j = 1...i$ whose invocation events appear before the response event of $\pi$ in $H_{\mathcal{L}}$, and*

*(b) for every complete operation $\mathcal{L}.\mathsf{append}(\rho) \in H_{\mathcal{L}}$, there exists a history $H'_{\mathcal{L}}$ that extends $H_{\mathcal{L}}$ such that, for every history $H''_{\mathcal{L}}$ that extends $H'_{\mathcal{L}}$, every complete operation $\mathcal{L}.\mathsf{get}()$ in $H''_{\mathcal{L}} \setminus H'_{\mathcal{L}}$ returns a sequence that contains $\rho$.*

Observe that the above definition is equivalent to the one given in [10, Definition 4].

A DLO is an ***eventually consistent Byzantine-tolerant DLO*** (BDLO), if it satisfies the next three properties:

- *Byzantine Completeness (BC)*: All the $\mathsf{get}()$ and $\mathsf{append}()$ operations invoked by correct clients eventually complete.
- *Byzantine Strong Prefix (BSP)*: If two *correct clients* issue two $\mathsf{get}()$ operations that return record sequences $S$ and $S'$ respectively, then either $S$ is a prefix of $S'$ or vice-versa.
- *Byzantine Eventual Consistency (BEC)*: This is the property of Definition 12 with respect to the $\mathsf{get}()$ operations invoked by correct clients and the $\mathsf{append}(r)$ operations that append the records $r$ returned in those $\mathsf{get}()$ operations.

## B Acronyms table

| | |
|---|---|
| DLT | Distributed Ledger Technologies |
| DLO | Distributed Ledger Object |
| SDLO | Smart Distributed Ledger Object |
| BDLO | Byzantine-tolerant Distributed Ledger Object |
| SBDLO | Smart Byzantine Distributed Ledger Object |
| G-Set | Grow-only Set |
| DSO | Distributed Grow-only Set Object |
| BDSO | Byzantine-tolerant Distributed Grow-only Set Object |
| BRB | Byzantine Reliable Broadcast |
| BToB | Byzantine Total-order Broadcast Service |
| CRDTs | Conflict-Free Replicated Data Type |

**Table 1** Meaning of acronyms