Atomic Appends in Asynchronous Byzantine Distributed Ledgers

Vicent Cholvi[a], Antonio Fernández Anta[b], Chryssis Georgiou[c], Nicolas Nicolaou[d], Michel Raynal[e], Antonio Russo[b,f]

[a] *Universitat Jaume I, Spain*
[b] *IMDEA Networks Institute, Spain*
[c] *Department of Computer Science, University of Cyprus, Nicosia, Cyprus*
[d] *Algolysis Ltd, Lemesos, Cyprus.*
[e] *IRISA, France & PolyU, Hong Kong*
[f] *Universidad Carlos III de Madrid, Spain*

Abstract

A Distributed Ledger Object (DLO) is a concurrent object that maintains a totally ordered sequence of records. In this work we formalize a *linearizable Byzantine-tolerant Distributed Ledger Object* (BDLO), which is a linearizable DLO where clients and servers processes may deviate arbitrarily from their intended behavior (i.e. they may be Byzantine). The proposed formal definition is accompanied by algorithms that implement BDLOs on top of an underlying Byzantine Atomic Broadcast service.

Then we develop a suite of algorithms, based on the previous BDLO implementations, that solve the *Atomic Appends* problem in the presence of asynchrony, Byzantine clients and Byzantine servers. This problem occurs when clients have a composite record (set of basic records) to append to different BDLOs, in such a way that either *each* basic record is appended to its BDLO (and this must occur in good circumstances), or *no* basic record is appended. Distributed algorithms are presented, which solve the *Atomic Appends* problem when the clients (involved in the *Atomic Appends*) and the servers (which

⋆A preliminary version of this work has appeared in the proceedings of the 16th European Dependable Computing Conference (EDCC 2020).

Email addresses: vcholvi@uji.es (Vicent Cholvi), antonio.fernandez@imdea.org (Antonio Fernández Anta), chryssis@ucy.ac.cy (Chryssis Georgiou), nicolas@algolysis.com (Nicolas Nicolaou), michel.raynal@irisa.fr (Michel Raynal), antonio@antoniorusso.me (Antonio Russo)

maintain the BDLOs) may be Byzantine. Finally we provide proof of concept implementations and an experimental evaluation of the presented algorithms.
Keywords: Atomic Appends, Asynchrony, Blockchain, Byzantine process, Cooperation, Distributed Ledger Object, Synchronization.

---

1. Introduction

There has been a great interest recently in the so-called crypto-technologies (e.g., blockchain systems [1]), and distributed ledger technology (DLT) in general [2], which are becoming very popular and are expected to have a high impact in multiple aspects of our everyday life. Although such a recent popularity is primarily due to the explosive growth of numerous crypto-currencies, there are many applications of this core technology that are outside the financial industry. These applications arise from leveraging various useful features provided by distributed ledgers: decentralized information management like symbolic name resolution [3], trustable and verifiable databases used for e-voting [4], immutable record keeping used for audit trails [5], and general benefits to systems' reliability and security given the intrinsic distributed nature of the technology. However, there are many different DLT-based systems, and new ones are proposed almost everyday. Hence, it is extremely unlikely that a single DLT will prevail. This is forcing the DLT community to accept that it is inevitable to come up with ways to make DLTs interconnect and interoperate.

In that direction, a formal definition of a reliable concurrent object, termed *Distributed Ledger Object* (DLO), which endeavours to convey the essential elements of the many (permissioned) DLTs, was proposed in [6]. In particular, a DLO maintains a sequence of records, and has only two 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.

Using the above-mentioned formalism, the study of systems formed by multiple DLOs that interact among each other was pursued in [7], where the *Atomic Appends problem* was introduced. In this problem, several clients have a "com-

posite" record (a set of semantically-linked "basic" records) to append, each basic record has to be appended to a corresponding DLO, and it must be guaranteed that either all basic records are appended to their respective 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. DLO_A is a ledger maintaining digital deeds and 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 DLO_A and record r_B is appended in DLO_B *or* no record is appended in the corresponding DLOs.

In the work presented in [7], the clients were assumed to be selfish and rational [8], and could have different incentives for the different outcomes. Additionally, any client could fail by crashing. The authors 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.

Contributions. While [6] and [7] assume that clients and servers can only fail by crashing, several DLT-based systems assume both some servers (e.g., miners) and some clients (e.g., users) can act maliciously. To this respect, this article presents implementations where *both* the clients and the servers can be Byzantine, i.e., it presents implementations of *Byzantine-tolerant* linearizable DLOs. More precisely, the following contributions are presented.

- A formalization of the *linearizable Byzantine-tolerant Distributed Ledger Object*, in short BDLO (Section 2).
- Algorithms (with their correctness proof) that implement a permissioned linearizable BDLO (Section 3) in an asynchronous setting (enriched with an underlying Byzantine Atomic Broadcast service) in which up to f servers

can be Byzantine, and (i) an unbounded number of clients can be Byzantine (Section 3.1), or (ii) only a bounded number t of clients can be Byzantine (Section 3.2). In the second case it is possible to prevent spurious records to be appended by Byzantine clients without adding a specific mechanism.

- A definition of the *Atomic Appends* problem in a system with Byzantine failures (Definition 1 in Section 4). The definition of Atomic Appends in [7] was for crash failures and has been adapted to Byzantine failures. In particular, since it is not possible to prevent a Byzantine client from appending its record, the safety requirement for Byzantine Atomic Appends is that a correct client appends its record only if all other records are appended.

- Algorithms (with their correctness proof) that combine BDLO implementations to solve the Atomic Appends problem (Section 4). We provide two solutions. (i) Following the Smart DLO presented in [7] for process crash failures, a Smart version of BDLO (SBDLO) is first introduced, which aggregates and coordinates the append of multiple records (Section 4.1). (ii) Then, it is shown how the problem can be solved by replacing the SBDLO with a "classical" BDLO and the use of a set of "helper" processes, a subset of which can be Byzantine (Section 4.2).

- We develop and evaluate proof of concept implementations of the unbounded, bounded, and smart BDLOs.

  - We demonstrate the feasibility of the proposed algorithms described in Section 3 and validate their correctness through an implementation based on Tendermint [9].

  - We show the advantages of leveraging an SBDLO for coordinating the Atomic Appends of multiple clients. Typical mechanisms based on hashlocks and timelocks, such as [10], used for implementing atomic swaps between ledgers, solve the problem in a response time which linearly increases with the number of involved ledgers. On the contrary, using the SBDLO keeps such time constant.

Note that the sequential specification of a DLO [6] requires that two clients issuing two GET operations, will return two record sequences S and S' such that either S is a prefix of S' or vice-versa. This property is called *strong prefix* in [11] and it essentially prevents forks, i.e., having more than one record sequence at any given time. In both [11] and [6] was shown that to implement such a property in a distributed setting, consensus is required. Following the definition of a DLO, our BDLO formalism also requires a strong prefix property. Therefore, our proposed BDLO implementations make use of a Byzantine Atomic Broadcast (BAB) service [12], which in turn is built on consensus algorithms [13]. BAB and consensus are computationally equivalent, so for our purposes it is more natural to use a BAB service instead of a consensus one, as BAB ensures total ordering of the messages exchanged; this property, together with additional machinery helps us in realizing the ordered sequence of records required by a BDLO. (In our experimental evaluation we have used Tendermint [9] to implement the BAB service.)

Related Work. One research direction derived from DLOs is exploring other object types that have weaker requirements than DLOs but can still be useful in given contexts. One example is the Distributed Grow-only Set object (DSO), which maintains a *set* of records instead of a sequence [14]. DSOs can be used with applications in which data persistence and fault-tolerance are required, but total order is not. Similarly, the Setchain abstraction has been proposed as an object type in between a DLO and a DSO [15]. A Setchain holds sets of records that are ordered among them via barrier operations.

The Atomic Appends problem is very related to the multi-party fair exchange problem [16], in which several parties exchange commodities so that everyone gives an item away and receives an item in return. However, the proposed solutions for this problem not only relies on computational heavy cryptographic technique [17, 18], but also assumes the existence of a centralized trusted party for the initial setup of the protocol. Such an assumption does not fit the decentralized scenarios in which distributed ledger are generally meant to be adopted.

Among the first problems identified involving the interconnection of DLTs was Atomic Cross-chain Swap [10, 19, 20], which can also be seen as a version of the fair exchange problem. In this case, two users want to exchange assets (usually cryptocurrency) in multiple blockchains. Van Glabbeek et al. [19] have given protocols to complete paths of cross-chain swaps. Herlihy [10] has formalized and generalized atomic cross-chain swaps beyond one-to-one paths, and shows how multiple cross-chain swaps can be achieved if the transfers form a strongly connected directed graph. To this respect, hashlocks and timelocks schemes were used, which require synchrony. Cross-chain swaps have been generalized to Cross-chain Deals [21, 22], in which multiple parties get involved in order to exchange assets. Cross-chain deals guarantee that all transfers are completed if all parties are correct (compliant with the protocol). However, if that is not the case, some assets may still be transferred, with the requirement that correct parties will not end up worse off than initially. In [19] it is shown that cross-chain swaps (and hence deals) require synchrony to be solved in bounded time. An alternative is to allow eventual termination condition, which can be done with partial synchrony and allowing users to abort [19, 22].

In most scenarios, atomic appends can be used to realize cross-chain deals, since every transfer in a deal can be a record/transaction to be appended in a ledger. However, by definition (cf. Definition 1), atomic appends do not guarantee termination in case of failures. On the other hand, they can be used beyond asset transfers, like changing an address which may involve appending records atomically in multiple ledgers (the postal service, utilities companies, insurance providers, etc.).

Unlike most DLT-based systems, in Hyperledger Fabric [23, 24] it is possible to have transactions that span several DLOs (called *channels* in Hyperledger Fabric). This allows solving the atomic cross-chain swap problem using a third trusted channel or a mechanism similar to two-phase commit [24]. Unfortunately, they are limited to the channels of a single and isolated Hyperledger Fabric deployment. There are other DLT-based systems that allow interactions between different DLOs, presumably with many more operations than atomic

swaps (e.g., Cosmos [25] or Polkadot [26]). These systems are based on their own protocols, so only DLOs of the same ecosystem can interact with each other.

Practical Byzantine Fault Tolerant (PBFT) [27] is a leader-based protocol that enables distributed systems to tolerate Byzantine faults. PBFT focuses on achieving consensus in order to reliably replicate a state machine across a network, ensuring fault tolerance and consistency. The PBFT *leader-based* consensus approach has been used as the basis of several permissioned BFT blockchain systems such as Hyperlegder [23], Tendermint [9], and SBFT [28]. Red Belly [29], a recent highly scalable and efficient BFT blockchain system deploys a *leaderless* consenus protocol, called Democratic BFT (DBFT) [30]. The creators of Red Belly argue that the leaderless nature of DBFT assists in the system's efficiency and scalability, as costly leader recovery mechanisms are no longer needed. Our BDLO specification (and implementation) is oblivious to the consensus mechanism used; Byzantine Atomic Broadcast [12] is used in a modular way to enable the ordering of the records in the ledger. This abstraction enables us to focus on specifying and proving correct the safety and liveness properties required for linearizable Distributed Ledgers when both clients and servers can be Byzantine.

2. Model and Definitions

Distributed Ledger Objects. A Distributed Ledger Object (DLO) is a concurrent object that stores a totally ordered sequence of *records* S (initially empty). A *record* is a triple $r = \langle \tau, p, v \rangle$, where p is the identifier of the process that created record r, v is the data of the record drawn from an alphabet Σ, and τ is a *unique* record identifier from a set $\mathcal{T}$ (e.g., the cryptographic hash of $\langle p, v \rangle$). Furthermore, a DLO $\mathcal{L}$ supports two operations, $\mathcal{L}.\text{APPEND}(r)$ and $\mathcal{L}.\text{GET}()$, which append a new record r to the sequence and return the whole sequence, respectively [6]. A DLO is oblivious to the syntax and semantics of the records it holds [6].

A DLO is implemented by a fixed set of *servers*, each one of them stor-

ing a copy of the sequence of records and running a distributed algorithm to collaborate with each other.

The DLO is used by a fixed set of *clients* that access it by invoking APPEND and GET operations, which are translated into request and response messages exchanged with the servers. An execution ξ of a DLO is a sequence of *invocation* and *return* events, starting with an invocation event.

An operation *op* is *complete* in an execution ξ if both the invocation and matching return of *op* appear in ξ. We also say that an operation *op precedes* an operation op', or op' *succeeds* *op*, in an execution ξ if the return event of *op* appears before the invocation event of op' in ξ; otherwise, the two operations are *concurrent* [31]. In this work, we focus on *linearizable* DLOs [6, Definition 2]. Informally, under linearizability, any APPEND or GET operation appears as if it occurs instantaneously at some point in the execution, yielding a total order among the operations. Such order must respect real-time ordering, and be consistent with the semantics of operations: no GET() preceding APPEND(r) returns a sequence with r, and all GET operations that succeed APPEND(r) do.

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

Failure Model. No message is lost, duplicated or modified. Processes (servers and clients) can fail arbitrarily, i.e., they can be Byzantine. Specifically, we assume a *Byzantine system* in which *up to f servers* can fail arbitrarily and that the total number of servers is at least $3f + 1$. For clients we consider two cases: (i) any number of clients can be Byzantine; (ii) up to t of at least $2t + 1$ clients can be Byzantine.

Public and private keys. We assume that each process p (client or server) has a pair of public and private keys, and a cryptographic certificate containing its public key. These certificates are generated by a reliable authority, so we discard the possibility of spurious or fake processes (there cannot be Sybil attacks),

and have been distributed to all the processes that may interact with each other. Hence, we also assume that the 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 [32]. Communication channels between correct processes are reliable but asynchronous.

Byzantine-tolerant DLOs. The first aim of this paper is to propose algorithms that implement a linearizable DLO $\mathcal{L}$ in a Byzantine asynchronous system. Here we present the properties that a DLO should satisfy with respect to *correct processes*, given that Byzantine processes may return any arbitrary sequence or append any arbitrary record:

- *Byzantine Completeness (BC)*: All the GET and APPEND operations invoked by correct clients eventually complete.
- *Byzantine Strong Prefix (BSP)*: If two *correct clients* 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.
- *Byzantine Linearizability (BL)*: Let C be all the complete operations issued by correct clients. Let $G \subseteq C$ be the set of all complete GET operations issued by correct clients. Let A be the set of complete APPEND operations $\mathcal{L}$.APPEND(r) such that $r \in S$ and S is the sequence returned by some operation $\mathcal{L}$.GET() $\in G$. Then linerizability holds with respect to the set of operations $C \cup A$ (notice that $r \in S$ implies that $\mathcal{L}$.APPEND(r) precedes $\mathcal{L}$.GET() in the total order on the operations). This property is similar to the one described in [33] for registers.

In the remainder, we say that a DLO is *Byzantine Tolerant*, termed BDLO, if it satisfies the properties BC, BSP, and BL in a Byzantine system.

Byzantine Atomic Broadcast. The algorithms presented below to implement BDLOs are based on an underlying Byzantine Atomic Broadcast (BAB) [34, 32, 12] service, which ensures total ordering of the messages exchanged. The BAB service has two operations: *BAB-broadcast*(m) used by a server to

broadcast a message m to all servers, and *BAB-deliver*(m) used by the BAB service to deliver a message m to a server.

Such a communication abstraction is usually based on appropriate Byzantine-tolerant consensus algorithms [6, 13] (recall that consensus is required in order to implement the strong prefix property of a DLO). Consensus in the Byzantine failure model requires $3f + 1$. Since in this paper DLOs are implemented on top of it, we "inherit" the $3f + 1$ requirement. We note, however, that except from the BAB service, for the upper lever implementation (i.e., our DLO algorithms) it would suffice to assume $2f + 1$ as the minimum number of servers.

From a user point of view, BAB is defined by the following properties.

- *Validity*: if a correct server BAB-broadcasts a message, it eventually BAB-delivers it.
- *Agreement*: if a correct server BAB-delivers a message, all correct servers will eventually BAB-deliver that message.
- *Integrity*: a message is BAB-delivered by a correct server at most once, and only if it was previously BAB-broadcast.
- *Total Order*: the messages BAB-delivered by the correct servers are totally ordered (i.e., if a correct server BAB-delivers message m before message m', every correct server BAB-delivers these messages in the same order).

The work in [6] uses an underlying crash-tolerant Atomic Broadcast (AB) service to implement a crash-tolerant DLO. Due to the very nature of Byzantine faults, replacing AB with BAB in the algorithm in [6] is not sufficient to produce Byzantine-tolerant upper layer algorithms [13]. For example, in the crash-tolerant DLOs, it is enough for a client to receive a response from a server. However, in the presence of Byzantine servers, a client cannot trust the response of a single server, thus, it needs to receive replies from multiple servers (e.g., the same response from at least $f + 1$ different servers).

Code 1 API to the operations of a BDLO $\mathcal{L}$, executed by Client p

```
1:  Init: c ← 0
2:  function L.GET( )
3:      c ← c + 1
4:      send request (c, p, GET) to at least 2f + 1 different servers
5:      wait responses (c, i, GETRESP, S) from f + 1 different servers
6:                                              carrying the same sequence S
7:      return S
8:  function L.APPEND(r)
9:      c ← c + 1
10:     send request (c, p, APPEND, r) to at least 2f + 1 different servers
11:     wait responses (c, i, APPENDRESP, ACK) from f + 1 different servers
12:     return ACK
```

3. Algorithms for Byzantine-tolerant DLOs

This section presents algorithms implementing Byzantine-tolerant DLOs. It first assumes that there is no bound on the number of clients that can fail (Section 3.1). However, if any set of clients can be Byzantine, then there is no way to prevent a client from appending a semantically invalid record (unless we assume that servers are *clairvoyants*, in the sense of detecting such records simply by checking them). Thus, assuming a bound t on the maximum number of Byzantine clients, Section 3.2 provides algorithms that implement DLOs in such a context. In that case, the invalid records are detected by requesting an APPEND operation to be issued by at least $t + 1$ clients (the fact that an operation is issued by any set of $t + 1$ clients, guarantees that at least one of these clients is correct).

3.1. Unbounded Number of Byzantine Clients

Client Algorithm. The algorithm executed by a client that invokes a GET or APPEND operation on a DLO $\mathcal{L}$ is presented in Code 1. An operation starts with the **invocation** (event) of the corresponding function in Code 1, and it ends when the matching **return** instruction is executed (return event). A Byzantine client p may not follow Code 1 (as it may behave arbitrarily) but still be able to append a record r in the ledger (assuming the messages it sends to do so are properly authenticated). So, some correct client may obtain, in the response

Code 2 Algorithm u-ByDL: Byzantine-tolerant DLO; Code for Server i. The operator $\oplus$ denotes concatenation of sequences.

```
 1: Init: S_i ← ∅
 2: receive (c, p, GET) from process p
 3:     BAB-broadcast(c, p, GET, i)
 4: upon (BAB-deliver(c, p, GET, j)) do
 5:     send response (c, i, GETRESP, S_i) to p
 6: receive (c, p, APPEND, r) from process p
 7:     BAB-broadcast(c, p, APPEND, r, i)
 8: upon (BAB-deliver(c, p, APPEND, r, j)) do
 9:     if (r ∉ S_i) then
10:         S_i ← S_i⊕r
11:         send resp. (c, i, APPENDRESP, ACK) to p
```

to a GET operation, a sequence that contains a record appended by Byzantine clients.

When an operation is invoked, a correct client increments a local counter and then sends operation requests to a set of at least $2f + 1$ servers, to guarantee that at least a correct server broadcasts it and that it will receive at least $f + 1$ responses. A GET operation completes when the client receives $f + 1$ *consistent* replies and an APPEND completes when the client receives $f + 1$ ACK from different servers. Both cases guarantee the response from at least one correct server.

Server Algorithm. The algorithm executed by the servers is presented in Code 2. It is denoted u-ByDL (for "unbounded Byzantine Distributed Ledger"). The algorithm uses the Byzantine Atomic Broadcast service to impose a total order in the messages shared among the servers. Operations received from clients are BAB-broadcast using this service, which are eventually BAB-delivered. The properties of the BAB service guarantee that all correct servers receive the same sequence of messages BAB-delivered, and hence process the operations at the same point, maintaining their states consistent. Figure 1 provides an example of how our ledger works.

Theorem 1. *Algorithm u-ByDL implements a linearizable Byzantine Tolerant DLO.*

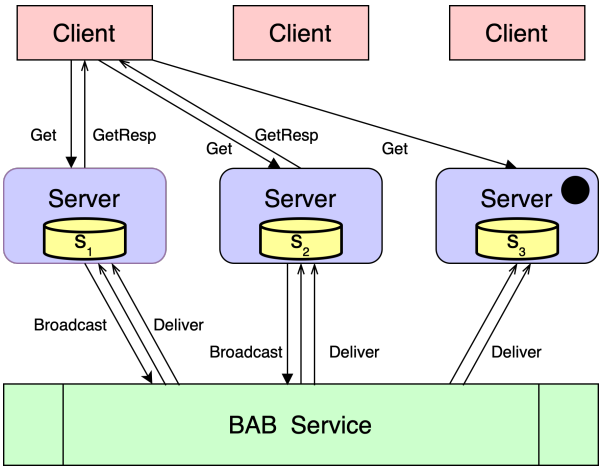


Figure 1: This drawing illustrates how the u-ByDL algorithm reacts to a GET operation issued by a client (the illustration for an APPEND operation is similar). There are three servers, one of them can be Byzantine (marked with a black dot) and the rest are correct (i.e., S_1 and S_2 are replicas of the same sequence). The client sends requests to three servers, and receives an answer from all the correct ones since in this case the Byzantine is behaving as a crashed node. It can be seen that, despite of the Byzantine server behavior, the client finally receives a correct answer from 2 servers, and, therefore, obtains a correct sequence. The outcome would have been the same even in the case in which the Byzantine server acted correctly but returning a different sequence in the attempt to cheat the client.

Proof. We have to show that any execution of the algorithm satisfies the properties BC, BSP and BL.

Liveness BC. The algorithm guarantees the liveness property BC with respect to the assumed Byzantine failure model. More precisely, each correct client sends requests to $2f + 1$ servers for an operation π and waits for $f + 1$ servers to reply. Given that the channels are reliable and up to f servers may fail (and thus may not reply), at least $f + 1$ correct servers eventually receive and BAB-broadcast the request from p. According to the BAB-Validity property, each message broadcasted by a correct server is eventually BAB-delivered. Furthermore, by the BAB-Agreement property, all correct servers eventually BAB-deliver the messages broadcast by correct servers. From the failure model there are at least $3f + 1$ servers, thus at least $f + 1$ correct servers eventually

will reply to operation π, and hence the client will receive at least $f + 1$ replies and terminate.

Safety BSP. Byzantine Strong Prefix requires that, if two GET operations from two correct clients return sequences S and S' resp., then either S is a prefix of S' or S' is a prefix of S. To derive contradiction let as assume that S is not a prefix of S'. Let $S = r_1 r_2 \dots r_n$ and $S' = r'_1 r'_2 \dots r'_m$, and that $m \geq n$ wlog. As S' is not a prefix of S, then $\exists\ r_i$ in S, for $1 \leq i \leq n$ s.t. $r_i \neq r'_i$. From the algorithm it follows that the GET operations received S and S' from at least one correct server as each get operations waits for $f + 1$ *different* servers to reply with the *same* sequence. Let s be a correct server that returned S and s' be a correct server that replied with S'. A correct server only appends a record to its ledger if it BAB-delivers it. Thanks to BAB-Integrity this means that a correct server BAB-Broadcasted it and according to BAB-Agreement if s BAB-delivers r_j then s' will BAB-deliver r_j as well. Furthermore, each record r_k, for $1 \leq k \leq j$, that has been BAB-delivered in s will also be BAB-delivered in s' and, according to the BAB-Total Order, those records will be delivered in the same order in both correct servers. This can be seen with a simple induction. The first record of S, r_1, will be BAB-delivered to both servers s and s' (by BAB-Agreement property). Record r_2 will be BAB-delivered after r_1 in s. By BAB-Agreement property r_2 will be BAB-delivered to s' as well and by BAB-Total Order r_2 cannot be delivered before r_1. So by the delivery of r_2 to s and s', both servers contain the sequence $r_1 r_2$. Suppose this is true up to record r_k, for $k < n$, i.e. both servers contain sequence $r_1 \dots r_k$ after the BAB-delivery of r_k. As noted before, record r_{k+1} will be BAB-delivered to both servers s and s' and by the total order property r_{k+1} cannot be delivered before any record r_j, with $j \leq k$. Thus, after the BAB-delivery of r_{k+1} both servers will contain the sequence $r_1 \dots r_k, r_{k+1}$. By the induction it follows that, after the BAB-delivery of r_n to both s and s', they contain sequences $r_1 \dots r_n$. This is sequence S. Furthermore any record r_m, for $m > n$, that is BAB-delivered to s' will be placed after r_n in its local sequence. Thus, S is a prefix of S' and that

contradicts our initial assumption.

Safety BL. Byzantine Linearizability requires that: (i) APPEND operations are ordered with respect to all other operations, (ii) if a GET operation returns a sequence that contains a record r_j then an APPEND(r_j) operation preceded that GET, and (iii) if a GET operation completes before the invocation of another GET operation, i.e. $\text{GET}_1 \rightarrow \text{GET}_2$, then GET_1 returns a sequence S that is a prefix of the sequence S' returned by GET_2. Total ordering of all operations is ensured by the BAB service. If π_1 happens before π_2, i.e. $\pi_1 \rightarrow \pi_2$ and both are executed by correct processes, clients send π_1 and π_2 messages to $2f + 1$ servers so at least $f + 1$ correct servers will receive and BAB-broadcast them. Thus, each correct server will eventually BAB-deliver those operations by BAB-Validity and BAB-Agreement properties. Moreover, thanks to BAB-Total Order each server processes π_1 (including the reply to the invoking client) in its sequence before receiving, and thus processing π_2. If π_1 and π_2 are both APPEND this proves (i) while if π_2 is a GET returning a sequence containing a record r, at least a correct server answered to the client including it, so that record could be present only if an APPEND happened before. Finally if both operations are GET there is no possibility for GET_1 to return a longer sequence than GET_2; so, from the BSP proof, sequence returned by GET_1 must be a prefix of sequence returned by GET_2 □

Recall that DLOs are oblivious to the syntax and semantics of the records they hold [6] (i.e., they do not care about the meaning of the records appended by Byzantine clients, which can be semantically invalid records).

3.2. Bounded Number of Byzantine Clients

To prevent spurious records from being added in the BDLOs, this section assumes that at most t clients can be Byzantine (let us recall that Sybil attacks are not possible). This is achieved by using a technique that is based on making several clients append the same record. It is hence assumed that a valid record r is appended by a set N of at least $2t + 1$ clients that invoke the operation APPEND(r) using Code 1 in parallel. The rationale behind this approach is that

Code 3 Algorithm b-ByDL: Byzantine-tolerant BDLO with bounded set N of Byzantine clients; Code for processing the APPEND operation at Server i

```
1:  Init: S_i ← ∅
2:  receive (c, p, APPEND, r) from process p ∈ N
3:      BAB-broadcast(c, p, APPEND, r, i)
4:  end receive
5:  upon (BAB-deliver(c, p, APPEND, r, j)) do
6:      if (r ∈ S_i) then send response (c, i, APPENDRESP, ACK) to p
7:      else
8:          if ((-, -, APPEND, r, -) has been received from a set C of t + 1 different clients) then
9:              S_i ← S_i ⊕ r
10:             send response (c, i, APPENDRESP, ACK) to all q ∈ C
```

the correct servers will only add a record provided it is requested by a certain number of clients (enough to guarantee that the record is correct). Although the assumption that a number of clients agree on appending the same record may appear artificial, in the next section we introduce a scenario where this is not only feasible, but necessarily required. Namely, in algorithm BADDL (Code 5) a number of servers used for realizing *atomic appends* are requested, at one point, to take the role of clients appending the same record, being necessary that at most t of them are Byzantine (there, we also show how that is guaranteed). In general, this agreement could take place within a cluster of clients, where the clients would need to coordinate in order to append data, e.g., on a BDLO. Since clients are supposed to be Byzantine, agreement on the record to APPEND should happen at least among $2t+1$ clients in order to ensure that at least $t+1$ trigger the APPEND operation.

Now, the processing of the append messages at the servers has to be changed as described in Code 3, yielding Algorithm b-ByDL (for "bounded Byzantine Distributed Ledger").

Theorem 2. *Algorithm b-ByDL implements a linearizable BDLO that only contains records appended by correct clients.*

Proof. To prove b-ByDL correctness, we need to show that it satisfies both the liveness and safety properties of a BDLO with the special requirement that any

record is appended by a correct client.

Livenesss BC: We prove that property BC is satisfied. With similar arguments as in the proof of Theorem 1 we can show that an append request issued by a correct client will be received by at least $f + 1$ correct servers, and hence each correct server will BAB-deliver it thanks to BAB-Validity and BAB-Agreement. In addition, since we assume that at least $2t + 1$ clients agreed on the record to append, then each correct server will receive at least $t + 1$ requests for r. Thus, correct servers will reply to each client the requested the append and hence each correct client will receive at least $f + 1$ replies and terminate.

Safety. Following the proof of Theorem 1 we can show that b-ByDL satisfies both BSP and BL properties. What remains to show is that any record appended in the ledger is sent by a correct client. This follows from the fact that at least $t + 1$ correct clients issue append requests for the same record r. Given that the communication channels are reliable, those messages will eventually be received by all correct servers. Since the servers wait to receive $t + 1$ append requests for record r, they ensure that at least one correct client requested r to be appended. Hence, any record on the DLO was appended by a correct client, which completes the proof. □

4. Byzantine Atomic Appends

This section formulates the Atomic Appends problem, termed *AtomicAppends*, that captures the properties we need to satisfy when multiple operations need to append *dependent* records on different BDLOs. Informally, *AtomicAppends* requires that either *all* records will be appended (each in the appropriate BDLO) or *none* will be appended to a BDLO. But as seen in Section 3, in the presence of Byzantine failures, it is impossible to prevent a faulty client from appending its record without coordination with the rest of clients. Hence, this makes *AtomicAppends* a non-trivial task.

In order to formally define the Atomic Appends problem, we first introduce some notation.

We say that 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 a BDLO, say $\mathcal{L}'$. Two records, r and r', are *mutually dependent*, if r depends on r' and r' depends on r.

Definition 1 (2-AtomicAppends). *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.*

As mentioned above, it is not possible to prevent a faulty client q from appending its record r_q even if the correct client p does not. What the safety property AAS guarantees is that the opposite cannot happen. This is analogous to the property in cross-chain deals [21] that a correct process cannot end up worse than at the beginning.

We say that an algorithm *solves* the 2-AtomicAppends problem under a given system, if it guarantees the safety and liveness properties AAS and AAL of Definition 1 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 only in one of the BDLOs (to prevent safety AAS).

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 up to k BDLOs.) From this point onwards, we will focus on the 2-AtomicAppends problem, and when clear from the context, we will refer to it simply as *AtomicAppends*.

Code 4 API for 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 SBDLO $\mathcal{L}$. Code for Client p.

1: **function** AtomicAppends$(p, \{p, q\}, r_p, \mathcal{L}_p, r_q))$
2: $\quad\mathcal{L}$.APPEND$(\langle \tau, p, v\rangle)$, where $v = \langle p, \{p, q\}, r_p, \mathcal{L}_p, r_q\rangle$
3: $\quad$**return** ACK
4: // Client p will know the Atomic Appends operation was completed successfully when it receives notifications from $t + 1$ different SBDLO servers. (See Code 5.)//

In the remainder of this section, we show how the previously introduced algorithms for implementing linearizable BDLOs can be combined and adapted to solve the Atomic Appends problem. First, we build a Smart BDLO (SBDLO) to aggregate and coordinate the append of multiple records (Sect. 4.1). Then, we show how the problem can be solved by replacing the SBDLO with a "classical" BDLO and the use of a set N of at least $2t + 1$ "helper" processes, of which at most t can fail (Sect. 4.2).

For simplicity, we first consider the 2-AtomicAppends problem, where two clients, p and q, attempt to append atomically two mutually dependent records r_p and r_q, in BDLOs $\mathcal{L}_p$ and $\mathcal{L}_q$, respectively. Furthermore, in the reminder we assume that BDLOs $\mathcal{L}_p$ and $\mathcal{L}_q$ use Algorithm b-ByDL to tolerate up to t Byzantine clients and f Byzantine servers, and only accept APPEND operations from a known set N of at least $2t + 1$ clients, of which at most t can fail.

4.1. Atomic Appends Using a Smart BDLO

As proposed in [7], in order to coordinate the individual appends we will use a Smart BDLO $\mathcal{L}$, that is a special BDLO to which clients p and q delegate the task of appending their records in the respective ledgers. They do that by appending in the SBDLO a description of the Atomic Appends operation to be completed, as shown in Code 4. Client p uses the APPEND operation to provide the SBDLO 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 SBDLO $\mathcal{L}$ is a BDLO with unbounded number of faulty clients but that *only allows the creator of a record to append it.* $\mathcal{L}$ is implemented with a set

Code 5 Algorithm BAADL: Smart Byzantine-tolerant SBDLO; Only the code for the APPEND operation is shown; Code for Server i

```
1:  Init: S_i ← ∅
2:  receive (c, p, APPEND, r) from process p
3:      BAB-broadcast(c, p, APPEND, r, i)
4:  upon (BAB-deliver(c, p, APPEND, r, j)) do
5:      if (r ∉ S_i) then
6:          S_i ← S_i ⊕ r
7:          send response (c, i, APPENDRESP, ACK) to p
8:          if (r.v = ⟨p, {p, q}, r_p, L_p, r_q⟩) and (∃r' ∈ S_i : r'.v = ⟨q, {p, q}, r_q, L_q, r_p⟩) then
9:              L_p.APPEND(r_p)
10:             L_q.APPEND(r_q)
11:             // Once the two append operations complete, the server will notify clients p and q
12:              that both records r_p and r_q have been appended to L_p and L_q, respectively. //
```

N of at least $2t + 1$ servers, out of which at most t may be Byzantine. Hence, the APPEND operation in the client side (Line 2 in Code 4) is implemented as described in Code 1, with t instead of f as the maximum number of faulty servers.

Code 5 describes the APPEND operation of Algorithm BAADL (from Byzantine Atomic Appends Distributed Ledger) that implements the SBDLO (the rest of the algorithm is as in Code 2). As expected, it is very similar to the implementation of a BDLO without restrictions in the number of Byzantine clients, but with a 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 $\mathcal{L}_q$. So, essentially, the servers implementing the SBDLO 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 SBDLO servers. In turn, the SBDLO 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 $t + 1$ different SBDLO servers.

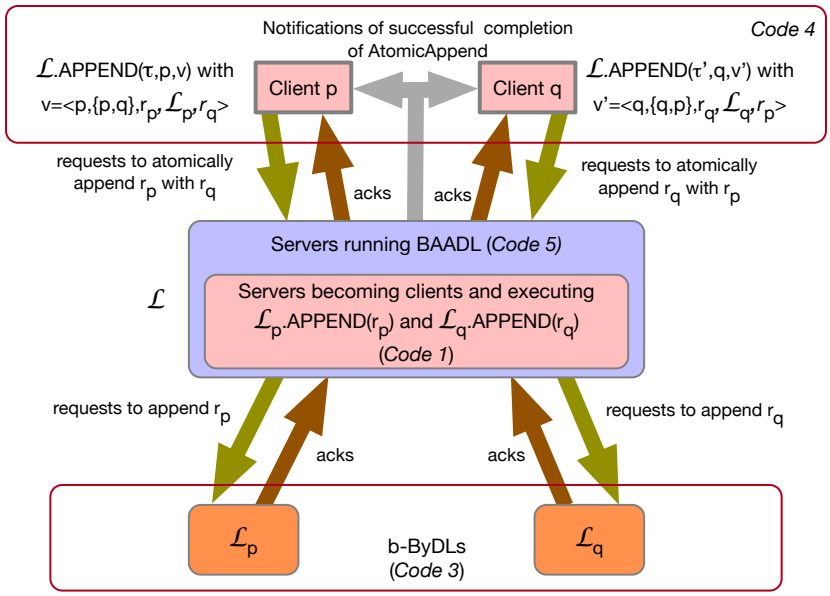


Figure 2: This drawing visualizes how the different algorithms involved in an atomic append interaction.

As mentioned above, each of the ledgers $\mathcal{L}_p$ and $\mathcal{L}_q$ are BDLOs with a known, bounded set N, of at least $2t + 1$ clients (which are the servers implementing the SBDLO $\mathcal{L}$), out of which at most t can be Byzantine. Each of these ledgers is implemented in a system of at least $3f + 1$ servers out of which at most f can be Byzantine, as presented in Algorithm b-ByDL (Code 3). Hence, a record is appended only if at least $t + 1$ clients from N issue append operations of the record. Notice that, differently from the case of ad-hoc clients, in the case of SBDLO at least $t + 1$ correct SBDLO servers will receive the requests by the external clients p and q and will issue the same APPEND operation in ledgers $\mathcal{L}_p$ and $\mathcal{L}_q$, making bounded BDLOs a practical system. Moreover, Line 2 of Code 3 is modified to verify that a client p attempting to append is in fact in the set N of authorized clients. Figure 2 illustrates how an *AtomicAppends* procedure works.

Theorem 3. *The combination of the API of Code 4 and the Algorithm BAADL*

solves the 2*-AtomicAppends problem.*

Proof. Let us first prove the liveness property AAL. Consider two correct clients p and q with records r_p and r_q, to be appended atomically in BDLOs $\mathcal{L}_p$ and $\mathcal{L}_q$, respectively. Since it is correct, p will eventually issue the call $\mathsf{AtomicAppends}(p, \{p, q\}, r_p, \mathcal{L}_p, r_q)$, which from Code 4 will trigger $\mathcal{L}.\text{APPEND}(\langle \tau, p, v \rangle)$, with $v = \langle p, \{p, q\}, r_p, \mathcal{L}_p, r_q \rangle$. From Code 1 (with t instead of f) and the process of the append messages in Algorithm BAADL, eventually all the correct servers i of the SBDLO will insert $\langle \tau, p, v \rangle$ in their sequences S_i. Similarly, eventually all the correct servers i of the SBDLO will insert $\langle \tau', q, v' \rangle$ with $v' = \langle q, \{q, p\}, r_q, \mathcal{L}_q, r_p \rangle$ in their sequences S_i.

Let us consider one such server i, and assume wlog that $\langle \tau, p, v \rangle$ is inserted fist in S_i. Then, as soon as $\langle \tau', q, v' \rangle$ is also inserted, the condition in Line 9 of Code 5 holds, and the operations $\mathcal{L}_p.\text{APPEND}(r_p)$ and $\mathcal{L}_q.\text{APPEND}(r_q)$ are issued. Since the SBDLO is implemented with at least $2t + 1$ servers out of which at most t are Byzantine, at least $t + 1$ servers will issue these APPEND operations. Hence, from Theorem 2, r_p and r_q will be appended to BDLOs $\mathcal{L}_p$ and $\mathcal{L}_q$, respectively.

We now prove the safety property AAS. Let us assume to reach a contradiction that AAS is not satisfied because, wlog, the record r_p of correct client p is appended in $\mathcal{L}_p$ while r_q is never appended in $\mathcal{L}_q$. Observe that $\mathcal{L}_p$ is a BDLO implemented with Algorithm b-ByDL, which requires at least $t + 1$ different clients appending the same record for the record to be in fact appended. There are two possibilities depending on who are these processes that append r_p in $\mathcal{L}_p$: they are (1) SBDLO servers or (2) they include processes that are not SBDLO servers. Let us consider each case separately.

In Case (1), there are at least $t + 1$ SBDLO servers that append r_p in $\mathcal{L}_p$. Then, at least one is correct, and does it by executing Line 10 in Code 5. Since all correct servers execute that line, and since there are at least $t + 1$ correct servers, record r_q is also appended in $\mathcal{L}_q$, which is a contradiction.

In Case (2), by assumption only the set N of servers of the SBDLO are

allowed to issue append operation in $\mathcal{L}_p$, and any append message sent by a process not in N will be rejected (recall that messages are authenticated). Hence, this case is not possible. □

Code 4 and the Algorithm BAADL are easily generalized to k-AtomicAppends. In Line 2 of Code 4 the client p sends the set of k clients appending records, and the $k-1$ records appended in addition to r_p. Similarly, in Line 9 of Code 5 the condition becomes that all k records to be appended are already in S_i. If so, all of them are appended in the k corresponding BDLOs.

4.2. Atomic Appends Using a Set of Helper Processes

While using a Smart BDLO solves the Atomic Appends problem as described above, it requires to implement a DLO that is aware of the contents of the records that are appended into it. In this section we describe how in fact the SBDLO can be replaced by a regular BDLO implemented with Algorithm u-ByDL (Code 2) and a set N of at least $2t+1$ helper processes, of which at most t can fail. The use of helper processes enables the use of BDLOs as "black boxes" in order to solve the Atomic Appends problem, making the solution transparent from the actual ledger implementation.

From the point of view of clients p and q, the new approach is transparent. Still they execute Code 4 to issue an Atomic Appends operation, with the difference that now ledger $\mathcal{L}$ is not "smart" anymore, but a regular Byzantine tolerant DLO (e.g., implemented with Code 2). Similarly, from the point of view of ledgers $\mathcal{L}_p$ and $\mathcal{L}_q$ the new approach is transparent, except that now their set N of legal clients to append in them is the set of helper processes described above.

The helper processes are continuously running a loop that monitors $\mathcal{L}$ for new Atomic Appends operations to complete. This process is described in Code 6. As can be seen there, a helper process h periodically issues a GET operation on $\mathcal{L}$ to obtain its latest contents. Then it checks if it contains pairs of matching Atomic Appends records that correspond to operations that have not been completed yet. (Observe that h maintains a set O_h of records from $\mathcal{L}$ that

Code 6 Algorithm used by a helper process to complete Atomic Appends operations; Code for process h

```
1: Init: O_h ← ∅
2: loop                                          ▷ Loop forever; execute loop body periodically
3:     S_h ← L.GET()
4:     while ∃r, r' ∈ S_h − O_h : r.v = ⟨p, {p, q}, r_p, L_p, r_q⟩ ∧ r'.v = ⟨q, {p, q}, r_q, L_q, r_p⟩ do
5:         L_p.APPEND(r_p)
6:         L_q.APPEND(r_q)
7:         O_h ← O_h ∪ {r, r'}
```

have been already used.) If so, it issues the corresponding APPEND operations to complete them. The proof that this approach solves the AtomicAppends problem is almost verbatim to the proof of Theorem 3, and it is omitted.

5. Proof of Concept Implementation and Evaluation

In this section we present and evaluate proof of concept implementations in the Go language of Byzantine-tolerant DLOs to show the feasibility of the proposed algorithms and to highlight the advantages of leveraging an SBDLO in coordinating clients for Atomic Appends.

5.1. Architecture and implementation choices

The BDLO algorithms presented above rely on a Byzantine Atomic Broadcast service that requires consensus in order to guarantee its properties. Our implementation is built on top of Tendermint [9] and its SDK [35] that provide a ready-to-use consensus layer for blockchains. In practice, Tendermint can be seen as an optimized version of a Byzantine Fault Tolerant consensus protocol that aggregates batches of values in a single consensus instance. The batch of values decided in each consensus instance is added to the Tendermint chain as a new block. In our case, consensus (block production) will happen as soon as a server invokes a BAB-Broadcast operation. BDLO servers will coincide with Tendermint consensus ones, since they are both implemented in the same executable.

Moreover the implementation optimizes the BAB-Broadcast and BAB-Delivery primitives avoiding to include in the Tendermint blockchain multiple

copies of the same client message (e.g., since a client sends the same request to many servers, Lines 10 and 4 of Code 1, BAB avoids to include in the blockchain a message it already saw in a block). This means that BAB is not fully a black box for DLO servers, but it cuts down the messages to deliver by at least a factor of $2f$ (only one message gets BAB-delivered instead of the $2f + 1$ that servers send per each append request). In a real scenario it also avoids that a Byzantine server floods the BAB service with copies of the same message.

On the client interaction side, servers expose a remote procedure call interface with the Append and Get operations. Both RPC clients and servers use the relative packages of the Golang standard library. Clients follow Code 1 to realise the Append operations; the workload is created by a benchmark orchestrator that assigns tasks to the clients' executables through the RabbitMQ message broker. As an optimization, the implementation of the Get request differs from Code 1 since it includes a parameter with the ledger height from which the server should return records. This avoids returning the whole sequence with each request. Append and Get operations can be treated as equivalent in terms of the load they incur, because of the similar algorithm (both operations trigger a BAB-Broadcast). Hence, we only present the performance of executing APPEND operations.

5.2. Metrics

We present the proof of concept performance in terms of the time APPEND operations take after invocation to successfully complete and return the control to the client. We will refer to this interval as *success time*. Note that clients wait for $f + 1$ responses before completing an operation (Lines 11 and 5 of Code 1).

We also present the *throughput* of operations that the BDLO system can sustain in terms of single managed APPEND requests. In the assumption that all clients are correct, like we have in our experiments, a single APPEND operation involves $2f + 1$ append requests. This metric is quite directly and uniquely influenced by the Tendermint consensus engine to agree on new blocks and expand their size.

5.3. Experiments

We remark that the experiments we report are only meant to provide a simple BDLO data structure implementation without any optimization, and show the differences in performance that each algorithm proposed implies under this simple implementation.

We use Amazon Web Services EC2 as our testing platform. We adopted two kinds of Virtual Machines: $m5$d.large for the BDLO servers, while $t2$.small has been chosen for running clients. Each server is run in a different m5d.large virtual machine. The clients are evenly distributed over 10 t2.small virtual machines. All machines were located in the same data center. Executable were compiled using Golang compiler 1.15.7 for linux/amd64.

In each experiment we create a BDLO with a fixed number of servers, and start a fixed number of concurrent clients that issue Append operations to the BDLO in a continuous loop, one after another without any delay. Each experiment was run 10 times. Between runs, servers and clients were stopped and their persistent state was removed in order to start each run with a clean system. In particular, at each run, the Tendermint blockchain that implements the BAB service is started from scratch (i.e. from the genesis block).

5.3.1. u-ByDL

The BDLO running the u-ByDL algorithm (Section 3.1, Code 2) can be treated as a baseline for the performance of our implementation. The operations issued by each client, GET or APPEND, do not need to wait for any other interaction by other clients, so the success time is mostly the overhead caused by the Tendermint consensus algorithm that implements the Byzantine Atomic Broadcast service. In fact the two bottom lines in Fig. 3(a) respectively represent the success time with a dummy BAB service (blue line), in order to isolate the network and processing delay, and the success time with a BAB service implemented on a single server (orange line). The other scenarios in this figure show that the median success time increases with the number of concurrent clients and the number of servers. For a fixed number of clients, the larger

the number of servers, the longer each consensus instance takes to be solved. For a fixed number of servers, the larger the number of clients, the higher the operations contention level at the servers.

In Fig. 3(b) the median throughput in number of operations per second is presented for each combination of number of servers and clients. In this figure, we do not observe the same drop in performance present in the success time figure, Fig. 3(a). This is because the batching of consensus values of the Tendermint implementation that is able to keep it constant across all the experiments, letting the block size to scale up when required. Anyway, once a peek of throughput around 200 clients is reached, all the three implementations running a consensus algorithm (4, 7, and 10 servers) saturate and start degrading performance around 300 clients.

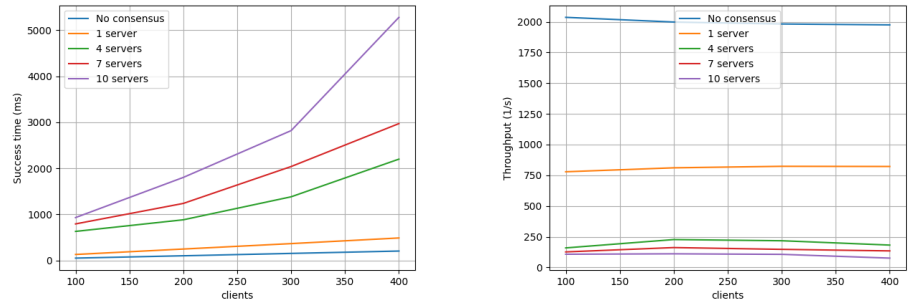
5.3.2. b-ByDL

The b-ByDL algorithm (Section 3.2, Code 3) differs from the previous one because it waits for a majority of clients to submit the exact same record before effectively appending it to the sequence. We observe in Fig. 3(c) that in this case the median success time drastically degrades in comparison to the u-ByDL algorithm since, even without failures, this BDLO needs to process $t + 1$ append requests of the same record before sending back an acknowledgement.

Regarding the throughput, as shown in Fig. 3(d) the behavior is similar to the u-ByDL case. In fact, records are packed in blocks by the Tendermint consensus engine and this happens independently from the fact that records are related to each other.

5.3.3. Atomic Appends

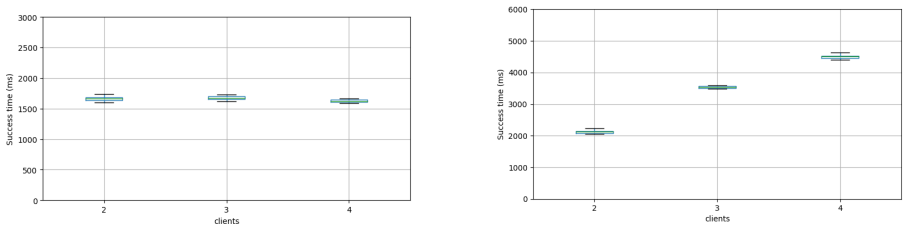
We now evaluate the performance of an implementation of the Byzantine Atomic Appends algorithm BAADL (Section 4.1, Code 5), in order to solve k-*AtomicAppends* for $k = 2, 3, 4$. The general architecture is formed of an SBDLO waiting for atomic append requests and 2 to 4 BDLOs. Once the atomic append requests are collected (Code 5, Line 8), the servers of the SBDLO act



(a) (b)

(c) (d)

Figure 3: Median values of the success time per operation on the BDLOs running the: (a) u-ByDL algorithm (Code 2), and (c) b-ByDL (Code 3) algorithm. Median values of the operation throughput for the BDLOs running the: (b) u-ByDL algorithm, and (d) b-ByDL algorithm.



(a) (b)

Figure 4: Success time distribution per Atomic Appends operation: (a) with a SBDLO and target BDLOs running the b-ByDL algorithm (algorithm BAADL, Code 5), and (b) using hashlocks and timelocks on BDLOs running the b-ByDL algorithm.

as clients for the target BDLOs. We ran scenarios with 2,3 and 4 clients issuing atomic appends on 2, 3, and 4 different BDLOs, respectively. In Fig. 4 (a) we can see how, independently from the number of clients and BDLOs involved, the average success time remains constant. This is because, in spite of the number of involved actors, our atomic append algorithm is always constituted of two communication rounds. In fact, the atomic append requests can be submitted in parallel since they do not require any interaction by the other clients and the same holds for the propagation of records to the target BDLOs since each BDLOs is completely independent from the others. By contrast, we also implemented, on top of our BDLOs, a simulation scenario in which the atomic appends solution is based on hashlocks and timelocks, as usually done to solve the problem of Atomic Swaps [10]. In this solution, with k BDLOs, first a smart contract with a hashlocks and a timelock is appended in each BDLO, and then the combination of each hashlock is appended in its corresponding BDLO. Each Append operation in this process can not be issued before the previous one is completed, in order to counter Byzantine behaviors. As a consequence, the success time per operation linearly depends on the number of involved clients and BDLOs (Fig. 4(b)).

6. Conclusion

The contributions of this work are a formalization of the notion of a linearizable Byzantine Tolerant Distributed Ledger Object (BDLO) and the design of algorithms implementing such objects in distributed settings where a subset of clients and servers may be Byzantine. A noteworthy feature of BDLOs lies in their ability to solve the Atomic Appends problem, where clients have a composite record (i.e., a set of mutually dependent basic records) to be appended, such that each basic record must be appended to a different BDLO, and either all basic records are appended or none.

The formalization of BDLOs requires a strong prefix property, which prevents the existence of more than one sequence at any point in time (i.e., no forks allowed, as termed in the blockchain parlance). As shown in [6, 11], this property requires consensus. Therefore, it would be interesting to investigate more relaxed (weaker) versions of this property (that might not require consensus) and study the guarantees that can be provided with the proposed framework. Another interesting direction would be to consider different data structures that do not require total order, such as the work in [14], which considers distributed Byzantine-tolerant grow-only sets.

Finally we presented a performance evaluation in order to benchmark (1) the impact of the different algorithms on the different kinds of BDLOs, and (2) the advantage of solving the Atomic Appends problem involving a third trusted party implemented with a BDLO.

Acknowledgment

This work has been partially supported by the Regional Government of Madrid (CM) grant EdgeData-CM – P2018/TCS4499 (cofunded by FSE & FEDER) and the Spanish Ministry of Science and Innovation grants ECID (PID2019-109805RB-I00) DiscoLedger (PDC2021-121836-I00) and PRX18/000163 (cofunded by FEDER). A preliminary version of this work has

appeared in the Proceedings of the 16th European Dependable Computing Conference (EDCC 2020).

References

[1] S. Nakamoto, Bitcoin: A peer-to-peer electronic cash system, https://bitcoin.org/bitcoin.pdf, [Online; accessed 22-February-2020] (2008).

[2] M. G. Zago, 50+ Examples of How Blockchains are Taking Over the World, Medium (2018).
URL https://medium.com/@matteozago/50-examples-of-how-blockchains-are-taking-over-the-world-4276bf488a4b

[3] Namecoin, Namecoin, https://www.namecoin.org/, [Online; accessed 22-February-2020].

[4] A. Russo, A. F. Anta, M. I. G. Vasco, S. P. Romano, Chirotonia: A scalable and secure e-voting framework based on blockchains and linkable ring signatures, in: Y. Xiang, Z. Wang, H. Wang, V. Niemi (Eds.), 2021 IEEE International Conference on Blockchain, Blockchain 2021, Melbourne, Australia, December 6-8, 2021, IEEE, 2021, pp. 417–424. doi:10.1109/Blockchain53845.2021.00065.
URL https://doi.org/10.1109/Blockchain53845.2021.00065

[5] T.-T. Kuo, H.-E. Kim, L. Ohno-Machado, Blockchain distributed ledger technologies for biomedical and health care applications, Journal of the American Medical Informatics Association 24 (6) (2017) 1211–1220.

[6] A. Fernández Anta, K. M. Konwar, C. Georgiou, N. C. Nicolaou, Formalizing and implementing distributed ledger objects, SIGACT News 49 (2) (2018) 58–76.

[7] A. Fernández Anta, C. Georgiou, N. Nicolaou, 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.

[8] M. J. Osborne, et al., An introduction to game theory, Vol. 3, Oxford university press New York, 2004.

[9] E. Buchman, Tendermint: Byzantine fault tolerance in the age of blockchains, Ph.D. thesis (2016).

[10] M. Herlihy, 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.

[11] E. Anceaume, A. D. Pozzo, R. Ludinard, M. Potop-Butucaru, S. Tucci Piergiovanni, Blockchain abstract data type, in: The 31st ACM on Symposium on Parallelism in Algorithms and Architectures, SPAA 2019, Phoenix, AZ, USA, June 22-24, 2019, ACM, 2019, pp. 349–358.

[12] Z. Milosevic, M. Hutle, A. Schiper, On the reduction of atomic broadcast to consensus with byzantine faults, in: SRDS 2011, 2011, pp. 235–244.

[13] M. Raynal, Fault-Tolerant Message-Passing Distributed Systems - An Algorithmic Approach, Springer, 2018. doi:10.1007/978-3-319-94141-7.
URL https://doi.org/10.1007/978-3-319-94141-7

[14] V. Cholvi, A. Fernández Anta, C. Georgiou, N. Nicolaou, M. Raynal, A. Russo, Byzantine-tolerant distributed grow-only sets: Specification and applications, in: V. Gramoli, M. Sadoghi (Eds.), 4th International Symposium on Foundations and Applications of Blockchain 2021, FAB 2021, May 7, 2021, University of California, Davis, California, USA (Virtual Conference), Vol. 92 of OASIcs, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, pp. 2:1–2:19. doi:10.4230/OASIcs.FAB.2021.2.
URL https://doi.org/10.4230/OASIcs.FAB.2021.2

[15] M. Capretto, M. Ceresa, A. Fernández Anta, A. Russo, C. Sánchez, Setchain: Improving blockchain scalability with byzantine distributed

sets and barriers, CoRR abs/2206.11845 (2022). arXiv:2206.11845, doi:10.48550/arXiv.2206.11845.
URL https://doi.org/10.48550/arXiv.2206.11845

[16] M. K. Franklin, G. Tsudik, Secure group barter: Multi-party fair exchange with semi-trusted neutral parties, in: R. Hirschfeld (Ed.), Financial Cryptography, Second International Conference, FC'98, Anguilla, British West Indies, February 23-25, 1998, Proceedings, Vol. 1465 of Lecture Notes in Computer Science, Springer, 1998, pp. 90–102.

[17] S. Micali, M. O. Rabin, J. Kilian, Zero-knowledge sets, in: 44th Symposium on Foundations of Computer Science (FOCS 2003), 11-14 October 2003, Cambridge, MA, USA, Proceedings, IEEE Computer Society, 2003, pp. 80–91.
URL http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=8767

[18] A. Mukhamedov, S. Kremer, E. Ritter, Analysis of a multi-party fair exchange protocol and formal proof of correctness in the strand space model, in: A. S. Patrick, M. Yung (Eds.), Financial Cryptography and Data Security, 9th International Conference, FC 2005, Roseau, The Commonwealth of Dominica, February 28 - March 3, 2005, Revised Papers, Vol. 3570 of Lecture Notes in Computer Science, Springer, 2005, pp. 255–269.

[19] R. van Glabbeek, V. Gramoli, P. Tholoniat, Cross-chain payment protocols with success guarantees, Distributed Computing (2023) 1–21.

[20] Y. Xue, D. Jin, M. Herlihy, Fault-tolerant and expressive cross-chain swaps, in: 24th International Conference on Distributed Computing and Networking, 2023, pp. 28–37.

[21] M. Herlihy, B. Liskov, L. Shrira, Adversarial cross-chain commerce, in: Principles of Blockchain Systems, Springer, 2021, pp. 133–154.

[22] M. Herlihy, B. Liskov, L. Shrira, Cross-chain deals and adversarial commerce, The VLDB journal 31 (6) (2022) 1291–1309.

[23] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. D. Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, S. Muralidharan, C. Murthy, B. Nguyen, M. Sethi, G. Singh, K. Smith, A. Sorniotti, C. Stathakopoulou, M. Vukolic, S. W. Cocco, J. Yellick, Hyperledger fabric: a distributed operating system for permissioned blockchains, in: R. Oliveira, P. Felber, Y. C. Hu (Eds.), Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018, ACM, 2018, pp. 30:1–30:15.
URL `http://dl.acm.org/citation.cfm?id=3190508`

[24] E. Androulaki, C. Cachin, A. D. Caro, E. Kokoris-Kogias, Channels: Horizontal scaling and confidentiality on permissioned blockchains, in: J. López, J. Zhou, M. Soriano (Eds.), Computer Security - 23rd European Symposium on Research in Computer Security, ESORICS 2018, Barcelona, Spain, September 3-7, 2018, Proceedings, Part I, Vol. 11098 of Lecture Notes in Computer Science, Springer, 2018, pp. 111–131.

[25] T. Inc., Cosmos, `https://cosmos.network`, [Online; accessed 22-November-2018].

[26] PolkaDot, PolkaDot, `https://polkadot.network`, [Online; accessed 22-November-2018].

[27] M. Castro, B. Liskov, Practical byzantine fault tolerance, in: Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation (OSDI), New Orleans, Louisiana, USA, February 22-25, 1999, USENIX Association, 1999, pp. 173–186.

[28] G. Golan Gueta, I. Abraham, S. Grossman, D. Malkhi, B. Pinkas, M. Reiter, D.-A. Seredinschi, O. Tamir, A. Tomescu, Sbft: A scalable and decentralized trust infrastructure, in: 2019 49th Annual IEEE/IFIP Interna-

tional Conference on Dependable Systems and Networks (DSN), 2019, pp. 568–580. doi:10.1109/DSN.2019.00063.

[29] T. Crain, C. Natoli, V. Gramoli, Red belly: A secure, fair and scalable open blockchain, in: 2021 IEEE Symposium on Security and Privacy (SP), 2021, pp. 466–483. doi:10.1109/SP40001.2021.00087.

[30] T. Crain, V. Gramoli, M. Larrea, M. Raynal, Dbft: Efficient leaderless byzantine consensus and its application to blockchains, in: 2018 IEEE 17th International Symposium on Network Computing and Applications (NCA), 2018, pp. 1–8. doi:10.1109/NCA.2018.8548057.

[31] L. Lamport, On interprocess communication, part I: Basic formalism, Distributed Computing 1 (2) (1986) 77–85.

[32] F. Cristian, H. Aghili, R. Strong, D. Dolev, Atomic broadcast: From simple message diffusion to byzantine agreement, Information and Computation 118 (1) (1995) 158 – 179.

[33] A. Mostéfaoui, M. Petrolia, M. Raynal, C. Jard, Atomic read/write memory in signature-free byzantine asynchronous message-passing systems, Th. Comp. Syst. 60 (4) (2017) 677–694.

[34] P. Coelho, T. C. Junior, A. Bessani, F. Dotti, F. Pedone, Byzantine fault-tolerant atomic multicast, in: DSN 2018, IEEE, 2018, pp. 39–50.

[35] Tendermint SDK Repository, `https://github.com/tendermint/tendermint`, [Online; accessed 16-July-2022].