Atomic Appends in Asynchronous Byzantine Distributed Ledgers

Vicent Cholvi[α] Antonio Fernández Anta[β] Chryssis Georgiou[γ] Nicolas Nicolaou[δ] Michel Raynal[ϵ]

[α] Universitat Jaume I, Spain [β] IMDEA Networks Institute, Spain
[γ] University of Cyprus, Cyprus [δ] Algolysis Ltd, Cyprus
[ϵ] IRISA, France & PolyU, Hong Kong

***Abstract*—A Distributed Ledger Object (DLO) is a concurrent object that maintains a totally ordered sequence of records, and supports two operations: APPEND, which appends a record at the end of the sequence, and GET, which returns the whole sequence of records. The work presented in this article is made up of two main contributions.**

The first contribution is a formalization of a *Byzantine-tolerant Distributed Ledger Object* (BDLO), which is a DLO in which 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.

The second contribution is 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 maintain the BDLOs) may be Byzantine.

***Index Terms*—Atomic Appends, Asynchrony, Blockchain, Byzantine process, Cooperation, Distributed Ledger Object, Synchronization.**

I. Introduction

Distributed Ledgers. There has been a great interest recently in the so-called crypto-technologies (e.g., blockchain systems [17]), and distributed ledger technology (DLT) in general [23], 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, such as a decentralized information management, immutable record keeping for possible audit trail, robustness, availability, security, and privacy (see, for instance, [12], [18], [21]. 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 DLTs, was proposed in [8]. 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 "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 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 Append 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 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.

Related Work. The Atomic Appends problem is very related to the multi-party fair exchange problem [9], 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 rely on cryptographic techniques [13], [16] and are not designed for distributed ledgers.

Among the first problems identified involving the interconnection of DLTs was Atomic Cross-chain Swap [10], [11], which can also be seen as a version of the fair exchange problem. In this case, two or more users want to exchange assets (usually cryptocurrency) in multiple blockchains. 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. Herlihy proves that the best strategy, in Game Theoretic sense, for the users is to follow the proposed algorithm. Unfortunately, these guarantees do not hold if the system is asynchronous.

Unlike most DLT-based systems, in Hyperledger Fabric [2], [3] 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 [3]. Additionally, these solutions do not require synchrony from the system. The ability of channels to access each other's state and interact is a very interesting feature of Hyperledger Fabric, in line with the techniques we assume from advanced distributed ledgers in this paper. Unfortunately, they seem to be limited to the channels of a given Hyperledger Fabric deployment.

There are other DLT-based systems under development that, like Hyperledger Fabric, intend to allow interactions between the different DLOs, presumably with many more operations than atomic swaps (e.g., Cosmos [22] or PolkaDot [19]). These systems are based on their own technology, so only DLOs in a given deployment can initially interact, other DLOs being connected via gateways.

Contributions. While [7] and [8] 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 *Byzantine-tolerant Distributed Ledger Object*, in short BDLO (Sect. II).
- Algorithms (with their correctness proof) that implement a linearizable BDLO (Sect. III) 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 (Sect. III-A), or (ii) only a bounded number t of clients can be Byzantine (Sect. III-B). 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 (Sect. IV).
- Algorithms (with their correctness proof) that combine BDLO implementations to solve the Atomic Appends problem (Sect. IV). We provide two solutions.
  - 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 (Sect. IV-A). The SBDLO is implemented with a set N of $n \geq 2t + 1$ servers up to which at most t can fail. The BDLOs on which the Atomic Appends is applied are implemented as BDLOs with a bounded number t of Byzantine clients, so it is guaranteed that only if at least one correct process in N appends in them, the append takes place.
  - Then, it is shown 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 up to t can be Byzantine (Sect. IV-B). These processes monitor (by periodically invoking GET operations) the BDLO for new Atomic Appends operations. Once "matching" Atomic Appends records are observed, the helper processes perform the APPEND operations to the corresponding BDLOs.

Note that the sequential specification of a DLO [8] 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 [1] and it essentially prevents forks, i.e., having more than one record sequence at any given time. In both [1] and [8] 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 [14], which in turn is built on consensus algorithms [20]. 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.

II. 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 [8].

A DLO is implemented by a fixed set of *servers*, each one of them storing 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*. In this work, we focus on *linearizable* DLOs [8]. 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 [6]. 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 G 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 $G \cup A$ (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 [15] 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. implement BDLOs are based on an underlying Byzantine Atomic Broadcast (BAB) [5], [6], [14] service, which ensures total ordering of the messages exchanged. Such a communication abstraction is based on appropriate Byzantine-tolerant consensus algorithms [8], [20]. Recall that consensus is required in order to implement the strong prefix property of a DLO (cf. [1], [8]).

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 message in the same order).

The work in [8] 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 algorithms [8] is not sufficient to produce Byzantine-tolerant upper layer algorithms [20].

III. 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 III-A). However, if all clients can be Byzantine, then there is no way to prevent a client from appending a meaningless 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 III-B provides algorithms that implement DLOs in such a context. In that case, the meaningless 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).

A. 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

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
```

Code 2 Algorithm u-ByDL: Byzantine-tolerant DLO; Code for Server i

```
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:      if ((c, p, GET, -) has been BAB-delivered f + 1 times
6:          from different servers) then send response (c, i, GETRESP, S_i) to p
7:  receive (c, p, APPEND, r) from process p
8:      BAB-broadcast(c, p, APPEND, r, i)
9:  upon (BAB-deliver(c, p, APPEND, r, j)) do
10:     if (r ∉ S_i) and ((c, p, APPEND, r, -) has been BAB-delivered from f + 1
11:         different servers) then
12:         S_i ← S_i‖r
13:         send resp. (c, i, APPENDRESP, ACK) to p
```

(assuming it is not meaningless). So, some correct client may obtain, in the response to a GET operation, a sequence that contains a meaningful records 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 $f+1$ correct servers receive it. A GET operation completes when the client receives $f+1$ *consistent* replies and an APPEND completes when the client receives $f+1$ replies 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. An operation is processed by a server only when it has been BAB-delivered $f+1$ times sent by different servers (hence at least one correct server sent it). 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.

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

Due to page limitation the proof of the liveness property LC is omitted. It can be found in the ArXiv tech report [4].

Safety BSP. Byzantine Strong Prefix requires that, if two GET operations from two correct clients return sequences S and S'

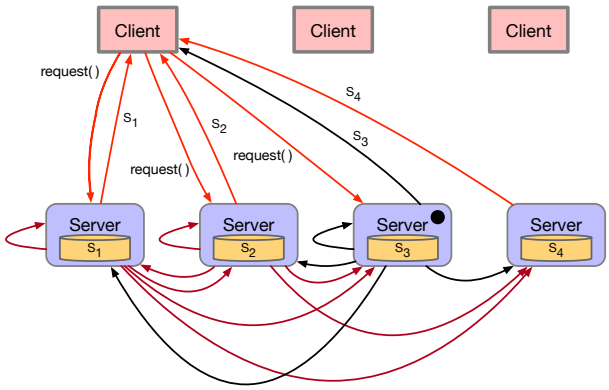

Fig. 1. This drawing illustrates how the u-ByDL algorithm reacts to a GET operation issued by a client (the illustration for an APPEND operation is quite similar). There are four servers, one of them is know to be Byzantine (marked with a black dot) and the rest are correct (i.e., S_1, S_2 and S_4 are replicas of the same sequence). The client sends requests to three servers, and receives an answer from all of them. It can be seen that, although the Byzantine server tries to cheat both the other servers and the client, finally the client receives a correct answer from 3 servers, and, therefore, obtains a correct sequence.

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 $S' = r'_1 r'_2 \dots r'_m$ with $m \geq n$. 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 the correct server that sent S and s' be the correct server that replied with S'. Before appending a record r_j in its local ledger, a correct server needs to wait for $f+1$ messages that contain r_j to be BAB-delivered. This guarantees that at least one correct server received the request for appending r_j and BAB-broadcast that record. According however to BAB-Agreement if s BAB-delivers r_j then s' will BAB-deliver r_j as well. Furthermore, for each record r_k, for $1 \leq k \leq j$, that is 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 $r_1 \dots r_n$. However 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. With similar reasoning we may show that if S is longer than S' then S' be a prefix of S.

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 returned by GET_2, say S'. The total ordering of the APPEND operations is ensured by the BAB service. In particular, if APPEND(r_1) happens before APPEND(r_2), i.e. APPEND(r_1) $\rightarrow$ APPEND(r_2), and both are executed by correct processes, then they will send the append message to $2f+1$ servers, out of which at least $f+1$ correct servers will receive and BAB-broadcast the append. Each correct server will BAB-deliver those appends by BAB-Validity and BAB-Agreement properties. Thus, each correct server will BAB-deliver at least $f+1$ messages for both records and will reply to the operations. Since APPEND(r_1) $\rightarrow$ APPEND(r_2) then there exists a correct server that replies to APPEND(r_1) before terminating. That server added r_1 in its sequence before receiving, and thus appending r_2. Therefore, by BAB-Total Order all the correct servers will append r_1 before r_2 in their local sequences, proving this way (i). As for point (ii) a GET operation obtains a sequence that contains a record r only if that sequence is received from at least a single correct server s. Thus, since s appended r in its sequence then an APPEND(r) operation must have executed before or concurrently with the GET operation. Finally, for two operations GET_1 and GET_2, s.t. $\text{GET}_1 \rightarrow \text{GET}_2$, it holds that the correct server, say s, that replies to GET_1 has delivered and replied to all APPEND operations with records in S_1 before BAB-delivering $f+1$ messages BAB-broadcast for GET_1. Since, $\text{GET}_1 \rightarrow \text{GET}_2$, the message for GET_2 will be BAB-delivered to s after the the delivery of the message of GET_1 at s. Since s is a correct server, then by BAB-Agreement and BAB-Total Order, all the correct servers will BAB-deliver the messages from GET_1 before the messages from GET_2. It holds also, that all the correct servers delivered all the records appended before GET_1, before the delivery of the messages from GET_2 as well. Thus, GET_2 will receive a sequence S' longer or the same size as S. From the proof of BSP though it follows that S is a prefix of S' and that completes the proof. ∎

At this point, we observe that DLOs are oblivious to the syntax and semantics of the records they hold [8] (i.e., they do not care about the meaning of the records appended by Byzantine clients, which can be correct or incorrect records.

B. 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 records. 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 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 DBLO.

Code 3 Algorithm b-ByDL: Byzantine-tolerant BDLO with bounded number 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
3:      BAB-broadcast(c, p, APPEND, r, i)
4:  upon (BAB-deliver(c, p, APPEND, r, j)) do
5:      if (r ∈ S_i) then send response (c, i, APPENDRESP, ACK) to p
6:      else
7:          if ((c, -, APPEND, r, -) has been BAB-delivered from f+1 different servers
8:                and 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
```

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 satisfies both liveness and safety properties of a BDLO with the special requirement that any record is appended by a correct client. Due to space limitations we only present here a sketch of the safety proof. Full proofs can be found in [4].

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 server 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. ∎

IV. 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 III, 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. A *Multi-Distributed Ledger Object* $\mathcal{M}$, termed MDLO, consists of a collection D of (heterogeneous linearizable) DLOs that support the following operations [7]: (i) $\mathcal{M}.\text{GET}_p(\mathcal{L})$, that returns the sequence of records $\mathcal{L}.S$, where $\mathcal{L} \in D$, and (ii) $\mathcal{M}.\text{APPEND}_p(\mathcal{L}, r)$, that appends the record r to the end of the sequence $\mathcal{L}.S$, where $\mathcal{L} \in D$. From the locality property of linearizability, it follows that a MDLO is linearizable, if it is composed of linearizable DLOs. *Multiple BDLOs*, termed MBDLO, are defined similarly over a collection of BDLOs (i.e., Byzantine-tolerant DLOs)[1].

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 of the property in atomic cross-chain swaps [10] 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 (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*.

[1]Note that we do not restrict whether the BDLOs in the MBDLO are implemented by common servers, or each BDLO is implemented by different servers, as long as the total number of servers and the bound on how many can fail is respected.

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, L_p, r_q))
2:     L.APPEND(⟨τ, p, v⟩), where v = ⟨p, {p, q}, r_p, L_p, r_q⟩
3:     return ACK
4: // Client p will know the Atomic Appends operation was completed successfully
5:  when it receives notifications from t + 1 different SBDLO servers. //
```

In the remainder of this section, we show how the previously introduced algorithms for implementing 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. IV-A). 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. IV-B).

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.

A. 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 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

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) and ((c, p, APPEND, r, -) has been
6:          BAB-delivered from t + 1 different servers) then
7:          S_i ← S_i‖r
8:          send response (c, i, APPENDRESP, ACK) to p
9:          if (r.v = ⟨p, {p, q}, r_p, L_p, r_q⟩) and
10:             (∃r' ∈ S_i : r'.v = ⟨q, {p, q}, r_q, L_q, r_p⟩) then
11:             L_p.APPEND(r_p); L_q.APPEND(r_q)
12:             // Once the above appends finish, the server will notify clients p and q
13:             that both records r_p and r_q have been appended to L_p and L_q,
14:             respectively. //
```

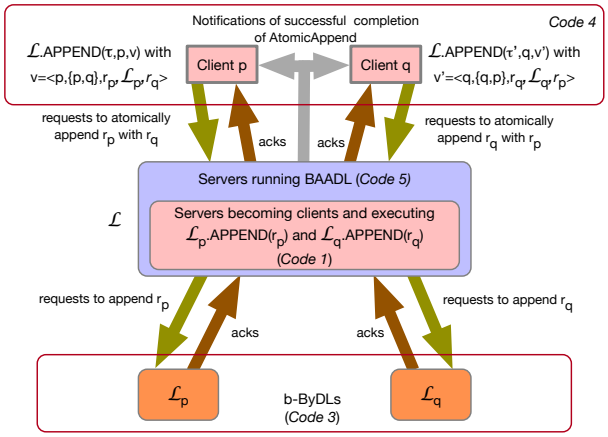

Fig. 2. This drawing visualizes how the different algorithms involved in an atomic append interact.

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.

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. These ledgers are implemented in a system of at least $2f+1$ 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, eventually p will issue the call AtomicAppends$(p, \{p, q\}, r_p, \mathcal{L}_p, r_q)$, which from Code 4 will trigger $\mathcal{L}$.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 $\mathcal{L}_p$.APPEND(r_p) and $\mathcal{L}_q$.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 11 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.

B. 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

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 :
5:         r.v = ⟨p, {p, q}, r_p, L_p, r_q⟩ ∧ r'.v = ⟨q, {p, q}, r_q, L_q, r_p⟩ do
6:         L_p.APPEND(r_p); L_q.APPEND(r_q)
7:         O_h ← O_x ∪ {r, r'}
```

u-ByDL (Code 2) and a set N of at least $2t + 1$ helper processes, of which at most t can fail.

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 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.

V. Conclusion

The contributions of this work are a formalization of the notion of a 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 [8], [1], 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.

Acknowledgment

This work was partially funded by the Spanish grant TIN2017-88749-R (DiscoEdge), the Region of Madrid EdgeData-CM program (P2018/TCS-4499), the NSF of China grant 61520106005, and by the Ministerio de Ciencia, Innovacion y Universidades grant PRX18/000163.

References

[1] Anceaume E., Del Pozzo A., Ludinard R., Potop-Butucaru M. nad Tucci Piergiovanni S., Blockchain Abstract Data Type, *Proc. 31st ACM Symposium on Parallelism in Algorithms and Architectures, (SPAA'19)*, ACM Press, pp. 349–358 (2019)

[2] Androulaki E. et al., Hyperledger fabric: a distributed operating system for permissioned blockchains, *Proceedings of the 13th EuroSys Conference*, ACM Press, pp. 30:1-30:15 (2018)

[3] Androulaki E., Cachin C., De Caro A., and Kokoris-Kogias E., Channels: horizontal scaling and confidentiality on permissioned blockchains, *Proc. 23rd ESORICS'18*, Springer LNCS 11098, pp. 111-131 (2018)

[4] Cholvi V., Fernández Anta, A., Georgiou Ch., Nicolaou N., and Raynal M., Atomic appends in asynchronous Byzantine distributed ledgers, arXiv:2002.11593 (2020)

[5] Coelho P.C., Ceolin Junior T., Bessani A., Dotti F.L., and Pedone F., Byzantine fault-tolerant atomic multicast. *Proc. 48th IEEE/IFIP Int'l Conference on Dependable Systems and Networks (DSN'18)*, IEEE Press, pp. 39-50 (2018)

[6] Cristian F., Aghili H., Strong R., and D. Dolev D., Atomic Broadcast: from simple message diffusion to Byzantine agreement, *Information & Computation*, 118(1)158-179 (1995)

[7] Fernández Anta A., Georgiou Ch., and Nicolaou N.C., Atomic appends: selling cars and coordinating armies with multiple distributed ledgers. *Proc. Int'l Conf. on blockchain Economics, Security and Protocols*, pp. 39-50 (2019)

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

[9] Franklin M.K and Tsudik G., Secure group barter: multi-party fair exchange with semi-trusted neutral parties, *Proc. 2d Int'l Conf. Financial and Cryptography (FC'98)*, pp. 90-102 (1998)

[10] Herlihy M.P., Atomic cross-chain swaps, *Proc. ACM Symposium on Principles of Distributed Computing (PODC'18)*, ACM Press, pp. 245-254 (2018)

[11] Herlihy M., Shrira L., Liskov B., Cross-chain Deals and Adversarial Commerce, *Proc. VLDB Endow.*, 13(2):100-113 (2019)

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

[13] Micali S., Rabin M.O., and Kilian J., Zero-knowledge sets, *Proc.44th Symposium on Foundations of Computer Science (FOCS'03)*, ACM Press pp. 80-91 (2003)

[14] Milosevic Z., Hutle M., and Schiper A., On the reduction of atomic broadcast to consensus with Byzantine faults, *Proc. 30th IEEE Symposium on Reliable Distributed Systems (SRDS'11)*, IEEE Press, pp. 235–244 (2011)

[15] Mostéfaoui A., Petrolia M., Raynal M. and Jard C., Atomic read/write memory in Byzantine asynchronous message-passing systems, *Theory of Computing Systems*, 60(4):677-694 (2017)

[16] Mukhamedov A. Kremer S, and Ritter E., Analysis of a multi-party fair exchange protocol and formal proof of correctness in the strand space model, Proc. 9th Int'l Conference on Financial Cryptography and Data Security, pp. 255-269 (2005)

[17] Nakamoto S., Bitcoin: A peer-to-peer electronic cash system. *Tech report*, bitcoin.org/bitcoin.pdf (2008)

[18] Namecoin, *Tech report*, https://www.namecoin.org/

[19] PolkaDot, *Tech report*,https://polkadot.network

[20] Raynal M., *Fault-tolerant message-passing distributed systems: an algorithmic approach*. Springer, 492 pages (2018)

[21] Spielman A., Blockchain: digitally rebuilding the real estate industry, *MS dissertation*, MIT (2016)

[22] Tendermint Inc., Cosmos, *Tech report*, https://cosmos.network

[23] Zago M.G., 50+ examples of how blockchains are taking over the world, *Tech report*, https://medium.com/@matteozago/50-examples-...-are-taking-over-the-world-4276bf488a4b (2018)