Brief Announcement: Implementing Byzantine Tolerant Distributed Ledger Objects

Vicent Cholvi
Universitat Jaume I, Spain
vcholvi@uji.es

Antonio Fernández Anta
IMDEA Networks Institute, Spain
antonio.fernandez@imdea.org

Chryssis Georgiou
University of Cyprus, Cyprus
chryssis@cs.ucy.ac.cy

Nicolas Nicolaou
Algolysis Ltd, Cyprus
nicolas@algolysis.com

Abstract

This work provides a proper formalization for Distributed Ledger Objects (as first defined in [1]), when processes may be Byzantine. The formal definitions are accompanied by algorithms to implement Byzantine Distributed Ledgers by utilizing a Byzantine Atomic Broadcast service.

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

Keywords and phrases Distributed Ledger Object, Byzantine Faults

Digital Object Identifier 10.4230/LIPIcs.DISC.2019.39

Category Brief Announcement

Funding This work was co-funded by the European Regional Development Fund and the Republic of Cyprus through the Research Promotion Foundation (Project: POST-DOC/0916/0090), by research funds from the University of Cyprus (CG-RA2019), 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 Spanish Ministerio de Educación Cultura y Deporte under grant PRX18/00163

1 Introduction

The work in [1] introduced the notion of a *Distributed Ledger Object* (DLO) in an attempt to provide a formalization of Distributed Ledgers (blockchains) from a Distributed Computing point of view. A DLO is a concurrent shared object that stores a totally ordered sequence of *records*, and supports two operations: *append* and *get*. A record can be seen as an abstraction of a transaction or a block of transactions. As operations may access the DLO concurrently, the work in [1] defines eventual, sequential, and linearizable consistency gurantees for DLOs. These formalisms were independent of the communication medium (message-passing or shared-memory) and the timing model (synchrony or asynchrony). Three DLO implementations, one for each consistency guarantee, were specified in [1] for a message-passing asynchronous model, assuming that clients and servers may crash. However, in existing blockchain systems, both the servers (e.g., miners) and the clients (e.g., users) could be acting maliciously. To this respect, in this work we propose implementations where *both* the clients and the servers can be Byzantine, i.e., we propose implementations of *Byzantine Tolerant* linearizable DLOs.

2 Model

Distributed Ledger Objects: A Distributed Ledger Object (DLO) is a concurrent object that stores a totally ordered sequence of records (initially empty). A DLO $\mathcal{L}$ supports two operations, $\mathcal{L}.append()$ and $\mathcal{L}.get()$, which append a new record to the sequence and return the whole sequence, respectively [1]. The DLO is implemented by a set of servers that collaborate running a distributed algorithm. The DLO is used by a 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 operation π is *complete* in an execution ξ, if both the request and matching response of π appear in ξ. We say that an operation π_1 *precedes* an operation π_2, or π_2 *succeeds* π_1, in an execution ξ if the response event of π_1 appears before the invocation event of π_2 in ξ; otherwise the two operations are *concurrent*.

Failure Model: In this work we assume that processes (servers and clients) can fail arbitrarily, i.e., we assume that failures are Byzantine. Hence, we assume a *Byzantine system* in which *any number of clients,* and *up to f servers* can fail arbitrarily. The total number of servers is at least $3f + 1$. 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 [3]. Communication channels between correct processes are reliable but asynchronous.

Byzantine-tolerant DLO: This paper aims to propose algorithms that implement a linearizable DLO $\mathcal{L}$ in Byzantine systems. Since Byzantine clients and server can behave arbitrarily, we define the properties that a DLO must satisfy adapted to Byzantine systems. In particular, since Byzantine processes may return any arbitrary sequence or append any record, the properties only consider the actions of *correct processes*.

- *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 $G \cup A$. This property is similar to the one described in [5] for registers.

In the remainder we say that a DLO is *Byzantine Tolerant* if it satisfies the properties BSP and BL in a Byzantine system. Observe that DLOs are oblivious to the syntax and semantics of the records they hold [1]. Hence, in this paper we do not need to care about whether the records appended by a Byzantine client are syntactically and semantically valid.

Byzantine Atomic Broadcast: In the algorithms we propose in this paper we use a Byzantine Atomic Broadcast (BAB) service for the server communication [2, 3, 4], that satisfies the following properties: validity, agreement, integrity and total order. Note that the work in [1] utilized a crash-tolerant Atomic Broadcast (AB) service to implement a crash-tolerant DLO. The properties assumed here for the BAB service are similar to their counterpart in the AB service, but applied only to correct processes. Despite the use of a BAB in this work, additional machinery is required in order to implement a Byzantine DLO and ensure the satisfaction of properties BSP and BL.

3 Algorithms for Byzantine-tolerant DLOs

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

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

Code 2 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 from different servers) then
6:         send resp. (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 at f + 1 different servers) then
11:        S_i ← S_i‖r
12:        send resp. (c, i, APPENDRESP, ACK) to p
```

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 when the corresponding function of Code 1 is invoked, and it ends when the matching **return** instruction is executed. 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. So, some correct client may obtain, in the response to a get operation, a sequence that contains r.

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 is completed when the client receives $f + 1$ *consistent* replies and an append is completed 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. 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). This implies that 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.

▶ **Theorem 1.** *The combination of the algorithms presented in Codes 1 and 2 implements a linearizable Byzantine Tolerant distributed ledger object.*

References

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

2 Paulo Coelho, Tarcisio Ceolin Junior, Alysson Bessani, Fernando Dotti, and Fernando Pedone. Byzantine fault-tolerant atomic multicast. In *DSN 2018*, pages 39–50. IEEE, 2018.

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

4 Zarko Milosevic, Martin Hutle, and André Schiper. On the reduction of atomic broadcast to consensus with byzantine faults. In *SRDS 2011*, pages 235–244, 2011.

5 Achour Mostéfaoui, Matoula Petrolia, Michel Raynal, and Claude Jard. Atomic read/write memory in signature-free byzantine asynchronous message-passing systems. *Th. Comp. Syst.*, 60(4):677–694, 2017.