# Atomic Appends: Selling Cars and Coordinating Armies with Multiple Distributed Ledgers

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

**Chryssis Georgiou**
Dept. of Computer Science, University of Cyprus, Nicosia, Cyprus
chryssis@cs.ucy.ac.cy

**Nicolas Nicolaou**
KIOS Research and Innovation CoE, University of Cyprus & Algolysis Ltd, Cyprus
nicolasn@ucy.ac.cy

## Abstract

The various applications using Distributed Ledger Technologies (DLT) or blockchains, have led to the introduction of a new “marketplace” where multiple types of digital assets may be exchanged. As each blockchain is designed to support specific types of assets and transactions, and no blockchain will prevail, the need to perform *interblockchain* transactions is already pressing.

In this work we examine the fundamental problem of interoperable and interconnected blockchains. In particular, we begin by introducing the *Multi-Distributed Ledger Objects* (MDLO), which is the result of aggregating multiple *Distributed Ledger Objects* – DLO (a DLO is a formalization of the blockchain) and that supports append and get operations of records (e.g., transactions) in them from multiple clients concurrently. Next we define the *AtomicAppends* problem, which emerges when the exchange of digital assets between multiple clients may involve appending records in more than one DLO. Specifically, AtomicAppend requires that either *all* records will be appended on the involved DLOs or *none*. We examine the solvability of this problem assuming *rational and risk-averse* clients that may *fail by crashing*, and under different client *utility* and *append* models, *timing models*, and client *failure scenarios*. We show that for some cases the existence of an intermediary is *necessary* for the problem solution. We propose the implementation of such intermediary over a specialized blockchain, we term *Smart DLO* (SDLO), and we show how this can be used to solve the AtomicAppends problem even in an asynchronous, client competitive environment, where all the clients may crash.

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

**Keywords and phrases** DLO, Interoperability, Atomic Appends, Rational Clients, Fault-tolerance

**Digital Object Identifier** 10.4230/OASIcs.Tokenomics.2019.3

**Funding** Partially supported by the Spanish Ministry of Science, Innovation and Universities grant DiscoEdge (TIN2017-88749-R), the Comunidad de Madrid grant EdgeData-CM (P2018/TCS-4499), the NSF of China grant 61520106005, and research funds from the University of Cyprus (CG-RA2019).

**Acknowledgements** We would like to thank Kishori Konwar, Michel Raynal, and Gregory Chockler for insightful discussions.

## 1 Introduction

Blockchain systems, cryptocurrencies, and distributed ledger technology (DLT) in general, are becoming very popular and are expected to have a high impact in multiple aspects of our everyday life. In fact, there is a growing number of applications that use DLT to support their operations [26]. However, there are many different blockchain systems, and new ones are proposed almost everyday. Hence, it is extremely unlikely that one single DLT or blockchain

International Conference on Blockchain Economics, Security and Protocols (Tokenomics 2019).
Editors: Vincent Danos, Maurice Herlihy, Maria Potop-Butucaru, Julien Prat, and Sara Tucci-Piergiovanni;
Article No. 3; pp. 3:1–3:15
OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

system will prevail. This is forcing the DLT community to accept that it is inevitable to come up with ways to make blockchains interconnect and interoperate.

The work in [7] proposed a formal definition of a reliable concurrent object, termed Distributed Ledger Object (DLO), which tries to convey the essential elements of blockchains. In particular, a DLO is a sequence of records, and has only two operations, append and get. The append operation is used to attach a new record at the end of the sequence, while the get operation returns the sequence.

In this work we initiate the study of systems formed by multiple DLOs that interact among each other. To do so, we define a basic problem involving two DLOs, that we call *the Atomic Append problem*. In this problem, two clients want to append new records in two DLOs, so that either both records are appended or none. The clients are assumed to be selfish, but rational and risk-averse [22], and may have different incentives for the different outcomes. Additionally, we assume that they may fail by crashing, which makes solving the problem more challenging. We observe that the problem cannot be solved in some system models and propose algorithms that solve it in others.

## 1.1 Related Work

The Atomic Append problem we describe above is very related to the multi-party fair exchange problem [8], in which several parties exchange commodities so that everyone gives an item away and receives an item in return. The proposed solutions for this problem rely on cryptographic techniques [18, 20] and are not designed for distributed ledgers. In this paper, as much as possible, we want to solve Atomic Appends on DLOs via their two operations append and get, without having to rely on cryptography or smart contracts.

Among the first problems identified involving the interconnection of blockchains was Atomic Cross-chain Swaps [13], 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. This problem can be solved by using escrows, hashlocks and timelocks: all assets are put in escrow until a value $x$ with a special hash $y = hash(x)$ is revealed or a certain time has passed. Only one of the users knows $x$, but as soon as she reveals it to claim her assets, everyone can use it to claim theirs. Observe that this solution assumes synchrony in the system, in the sense that timelocks assume that the time to claim an asset is bounded and known, and that timeouts can be used to detect crashes.

This technique was originally proposed in on-line fora for two users [1], and it has been specified, validated, adapted, and used [17, 21]. For instance, the Interledger system [11] will use a generalization of atomic swaps to transfer (and exchange) currency in a network of blockchains and connectors, allowing any client of the system to interact with any other client. The Lightning network [19, 23] also allows transfers between any two clients via a network of micro-payment channels using a generalized atomic swap. Both Interledger and Lighting route and create one-to-one transfer paths in their respective networks. Herlihy [13] 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, and that someone that follows it will never end up worst than at the start.

Unlike in most blockchain systems, in Hyperledger Fabric [5, 6] it is possible to have transactions that span several blockchains (blockchains are called *channels* in Hyperledger Fabric). This allows solving the atomic cross-chain swap problem using a third trusted channel or a mechanism similar to a two-phase commit [6]. 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, very 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 blockchain systems under development that, like Hyperledger Fabric, will allow interactions between the different chains, presumably with many more operations than atomic swaps. Examples are Cosmos [2] or PolkaDot [4]. These systems will have their own multi-chain technology, so only chains in a given deployment can initially interact, and other blockchain will be connected via gateways. Another proposal for interconnection of blockchains is Tradecoin [12], whose target is to interconnect all blockchains by means of gateways, trying to reproduce the way Internet works. Since the gateways will be clients of the blockchains, the functionality of the global interledger system will be limited by what can be done from the edge of the blockchains (i.e., by the blockchains' clients).

The practical need of blockchain systems to access the outside world to retrieve data (e.g., exchange rates, bank account balances) has been solved with the use of *blockchain oracles*. These are relatively reliable sources of data that can be used inside a blockchain, typically in a smart contract. The weakest aspect of blockchain oracles is trust, since the outcome or actions of a smart contract will be as reliable as the data provided by the oracle. As of now, it seems there is no good solution for this trust problem, and blokchains have to rely on oracle services like Oraclize [3].

## 1.2 Contributions

As mentioned above, in this paper we extend the study of the distributed ledger reliable concurrent object DLO started in [7] to systems formed of several such objects. Hence, the first contribution is the definition of the Multiple DLO (MDLO) system, as the aggregation of several DLOs (in similar way as a Distributed Shared Memory is the aggregation of multiple registers [25]). The second contribution is the definition of a simple basic problem in MDLO systems: the *2-AtomicAppends problem*. In this problem, the objective is that two records belonging to two different clients are appended to two different DLOs atomically. Hence, either both records are appended or none is. Of course, this problem can be generalized in a natural way to the $k$-Atomic Appends problem, involving $k$ clients with $k$ records and up to $k$ DLOs.

Another contribution, in our view, is the introduction of a crash-prone risk-averse rational client model, which we believe is natural and practical, especially in the context of blockchains. In this model, clients act selfishly trying to maximize their utility, but minimizing the risk of reducing it. We consider that this behavior is not a failure, but the nature of the client, and any algorithm proposed under this model (e.g., to solve the 2-AtomicAppends problem) must guarantee that clients will follow it, because their utility will be maximized without any risk. For a complete specification of the clients' rationality their utility function has to be provided. Two utility models are proposed. In the *collaborative utility model*, both clients want the records to be appended over any other alternative. In the *competitive utility model* a client still wants both records appended, but she prefers that only the other client appends. This client model is complemented with the possibility that clients can fail by crashing.

We explore hence the solvability of 2-AtomicAppends in MDLO systems in which the DLOs are reliable but may be asynchronous, and the clients are rational but may fail by crashing. The first results we present consider a system model in which clients do not crash, and show that Collaborative 2-AtomicAppends can be solved even under asynchrony, while Competitive 2-AtomicAppends cannot be solved. Then, we further study Collaborative

2-AtomicAppends if clients can crash. In the case that at most one of the two clients can crash, we show that, if each client must append its own record (what we call *no delegation*), Collaborative 2-AtomicAppends cannot be solved even under synchrony. This justifies exploring the possibility of *delegation:* any client can append any record, if she knows it. We show that in this case Collaborative 2-AtomicAppends can be solved, even if the system is asynchronous (termination is only guaranteed under synchrony, though). However, delegation is not enough if both clients can crash, even under synchrony. (See Table 2 for an overview.)

The negative results (for Competitive 2-AtomicAppends even without crash failures and for Collaborative 2-AtomicAppends with up to 2 crashes) justifies exploring alternatives to appending directly or delegating among clients. Hence, we propose the use of an entity, external to the clients, that coordinates the appends of the two records. In fact, this entity is a special DLO with some level of intelligence, which we hence call *Smart DLO* (SDLO). The SDLO is by design a reliable entity to which clients can delegate (via appending in the SDLO) the responsibility of appending their records to their respective DLOs when convenient. The SDLO hence collects all the records from the clients and appends them. Since the SDLO is reliable, all the appends will complete. If some record is missing, the SDLO issues no append, to guarantee the properties of the 2-AtomicAppends problem. Thus, the SDLO can be used to solve Competitive and Collaborative $k$-AtomicAppends even when all clients can crash.

We believe that SDLO opens the door to a new type of interconnection and interoperability among DLOs and blockchains. While the use of oracles to access external information in a smart contract (maybe from another blockchain) is widely known, we are not familiar with blockchain systems in which one blochchain (i.e., possibly a smart contract) issues transactions in another blockchain. We believe this is a concept worth to be explored further.

The rest of the paper is structured as follows. The next section describes the model used and defines the AtomicAppends problem. Section 3 explores the 2-AtomicAppends problem when clients cannot crash. Section 4 studies the 2-AtomicAppends problem when clients can crash but SDLOs are not used. Section 5 introduces the SDLO and shows how it solves the AtomicAppends problem. Finally, Section 6 presents conclusions and future work.

## 2 Problem Statements and Model of Computation

### 2.1 Objects and Histories

An object type $T$ is defined over the domain of values that any object of type $T$ may take, and the operations that any object of type $T$ supports. An object $O$ of type $T$ is a *concurrent object* if it is a shared object accessed by multiple processes [24]. A *history* of operations on an object $O$, denoted by $H_O$, is the sequence of operations invoked on $O$. Each operation $\pi$ contains an *invocation* and a matching *response* event. Therefore, a *history* is a sequence of invocation and response events, starting with an invocation. We say that an operation $\pi$ is *complete* in a history $H_O$, if the history contains both the invocation and the *matching* response events of $\pi$. History $H_O$ is *complete* if it only contains complete operations. History $H_O$ is *well-formed* if no two invocation events that do not have a matching response event in $H_O$ belong to the same process $p$. That is, each process $p$ invokes one operation at a time. An object history $H_O$ is *sequential*, if it contains a sequence of alternating invocation and matching response events, starting with an invocation and ending with a response. We say that an operation $\pi_1$ *happens before* an operation $\pi_2$ in a history $H_O$, denoted by $\pi_1 \rightarrow \pi_2$, if the response event of $\pi_1$ appears before the invocation event of $\pi_2$ in $H_O$.

**The Ledger Object (LO).** A *ledger* $\mathcal{L}$ (as defined in [7]) is a concurrent object that stores a totally ordered sequence $\mathcal{L}.S$ of *records* and supports two operations (available to any process $p$): (i) $\mathcal{L}.\mathsf{get}_p()$, and (ii) $\mathcal{L}.\mathsf{append}_p(r)$. A *record* is a triple $r = \langle \tau, p, v \rangle$, where $p$ is the identifier of the process that created record $r$, $\tau$ is a *unique* record identifier from a set $\mathcal{T}$, and $v$ is the data of the record drawn from an alphabet $\Sigma$. We will use $r.p$ to denote the id of the process that created record $r$; similarly we define $r.\tau$ and $r.v$. A process $p$ invokes an $\mathcal{L}.\mathsf{get}_p()$ operation to obtain the sequence $\mathcal{L}.S$ of records stored in the ledger object $\mathcal{L}$, and $p$ invokes an $\mathcal{L}.\mathsf{append}_p(r)$ operation to extend $\mathcal{L}.S$ with a new record $r$. Initially, the sequence $\mathcal{L}.S$ is empty.

▶ **Definition 1** (Sequential Specification of a LO [7]). *The* sequential specification *of a ledger $\mathcal{L}$ over the sequential history $H_{\mathcal{L}}$ is defined as follows. The value of the sequence $\mathcal{L}.S$ of the ledger is initially the empty sequence. If at the invocation event of an operation $\pi$ in $H_{\mathcal{L}}$ the value of the sequence in ledger $\mathcal{L}$ is $\mathcal{L}.S = V$, then:*

1. *if $\pi$ is an $\mathcal{L}.\mathsf{get}_p()$ operation, then the response event of $\pi$ returns $V$, while the value of $\mathcal{L}.S$ does not change, and*
2. *if $\pi$ is an $\mathcal{L}.\mathsf{append}_p(r)$ operation (and $r \notin V$), then at the response event of $\pi$ the value of the sequence in ledger $\mathcal{L}$ is $\mathcal{L}.S = V \| r$ (where $\|$ is the concatenation operator).*

In this paper we assume that ledgers are *idempotent*, therefore a record $r$ appears only once in the ledger even when the same record $r$ is appended to the ledger by multiple append operations (and hence the $r \notin V$ in the definition above).

## 2.2 Distributed Ledger Objects (DLO) and Multiple DLOs (MDLO)

**Distributed Ledger Objects (DLO).** A *Distributed Ledger Object (DLO)* $\mathcal{DL}$, is a concurrent LO that is *implemented* by (and possibly replicated among) a set $\mathcal{S}$ of (possibly distinct and geographically dispersed) computing devices, we refer as *servers*. Like any LO, $\mathcal{DL}$ supports the operations $\mathsf{get}()$ and $\mathsf{append}()$. We refer to the processes that invoke the $\mathsf{get}()$ and $\mathsf{append}()$ operations on $\mathcal{DL}$ as *clients*.

Each server $s \in \mathcal{S}$ may fail. Thus, the distribution and replication of $\mathcal{DL}$ offers availability and survivability of the ledger in case a subset of servers fail. At the same time, the fact that multiple clients invoke $\mathsf{append}()$ and $\mathsf{get}()$ requests to different servers, raises the challenge of *consistency*: what is the latest value of the ledger when multiple clients access the ledger concurrently? The work in [7] defined three consistency semantics to explain the behavior of $\mathsf{append}()$ and $\mathsf{get}()$ operations when those are invoked concurrently by multiple clients on a single DLO. In particular, they defined *linearizable* [14,16], *sequential* [15], and *eventual* [9] consistent DLOs. In this work we will focus on *linerizable* DLOs which according to [7] are defined as follows:

▶ **Definition 2** (Linearizable Distributed Ledger Object [7]). *A distributed ledger $\mathcal{DL}$ is* linearizable *if, given any complete, well-formed history $H_{\mathcal{DL}}$, there exists a sequential permutation $\sigma$ of the operations in $H_{\mathcal{DL}}$ such that:*

1. *$\sigma$ follows the sequential specification of a ledger object (Definition 1), and*
2. *for every pair of operations $\pi_1, \pi_2$, if $\pi_1 \rightarrow \pi_2$ in $H_{\mathcal{DL}}$, then $\pi_1$ appears before $\pi_2$ in $\sigma$.*

**Multiple DLOs (MDLO).** A *Multi-Distributed Ledger Object* $\mathcal{MDL}$, termed MDLO, consists of a collection $D$ of (heterogeneous) DLOs and supports the following operations: (i) $\mathcal{MDL}.\mathsf{get}_p(\mathcal{DL})$, and (ii) $\mathcal{MDL}.\mathsf{append}_p(\mathcal{DL}, r)$. The get returns the sequence of records $\mathcal{DL}.S$, where $\mathcal{DL} \in D$. Similarly, the append operation appends the record $r$ to the end of the sequence $\mathcal{DL}.S$, where $\mathcal{DL} \in D$. From the locality property of linearizability [14] it follows that a MDLO is linearizable, if it is composed of linearizable DLOs. More formally:

▶ **Definition 3** (Linearizable Multi-Distributed Ledger Object). *A multi-distributed ledger* $\mathcal{MDL}$ *is* linearizable *if* $\forall \mathcal{DL} \in D$, $\mathcal{DL}$ *is linearizable, where* $D$ *is the set of DLOs* $\mathcal{MDL}$ *contains.*

For the rest of this paper, unless otherwise stated, we will focus on MDLOs consisting of two DLOs. The same techniques can be generalized in MDLOs with more than two DLOs. In particular, we consider the records of two clients, $A$ and $B$, on two different DLOs. For convenience we use DLO$_X$ to denote the DLO appended by records from $X$, for $X \in \{A, B\}$. Similarly we denote as $r_X$ the record that $X \in \{A, B\}$ wants to append on $DLO_X$. Furthermore, we view the DLOs and MDLOs as black boxes that reliably implement the specified service, without going into further implementation details.

## 2.3 AtomicAppends: Problem Definition

Multi-DLOs allow clients to interact with different DLOs concurrently. This is safe when the records involved in concurrent operations are independent. However, it may raise semantic consistency issues when there exists inter-dependent records, e.g. a record $r_A$ must be inserted in DLO$_A$ when a record $r_B$ is inserted in DLO$_B$ and vice versa. More formally, we say that a record $r$ *depends* on a record $r'$, if $r$ may be appended on its intended DLO, say $\mathcal{DL}$, only if $r'$ is appended on a DLO, say $\mathcal{DL}'$. Two records, $r$ and $r'$, are *mutually dependent*, if $r$ depends on $r'$ and $r'$ depends on $r$. In this section we define a new problem, we term *AtomicAppends*, that captures the properties we need to satisfy when multiple operations attempt to append dependent records on different DLOs.

▶ **Definition 4** (2-AtomicAppends). *Consider two clients, A and B, with mutually dependent records* $r_A$ *and* $r_B$. *We say that records* $r_A$ *and* $r_B$ *are* appended atomically *on* $DLO_A$ *and* $DLO_B$ *respectively, when:*

- *Either both or none of the records are appended to their respective DLOs (safety)*
- *If neither A nor B fail, then both records are appended eventually (liveness).*

An algorithm *solves* the 2-AtomicAppends problem under a given system model, if it guarantees the safety and liveness properties of Definition 4.

The *k-AtomicAppends* problem, for $k \geq 2$, is a generalization of the 2-AtomicAppends that can be defined in the natural way ($k$ clients, with $k$ records, to be appended to up to $k$ DLOs.) 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*.

## 2.4 Communication, Timing and Append Models

The previous subsections are independent of the communication medium, and the failure and timing model. We now specify the communication and timing assumptions considered in the remainder of the paper. We also consider different models on who can append a specific record.

**Communication model:** We assume a *message-passing* system where messages are neither lost nor corrupted in transit. This applies to both the communication among clients and between clients and DLOs (i.e, the invocation and response messages of the operations).

**Timing models:** We consider *synchronous* and *asynchronous* systems with respect to both computation and communication. In the former, the evolution of the system is governed by a global clock and a local computation, a message delivery or a DLO operation is guaranteed to complete within a predefined time-frame. For simplicity, we set this time-frame to correspond to one unit of time. In the latter, no timing assumptions are made beyond that they will complete in a finite time.

**Append models:** We consider three different append models. In the first, and most restrictive one, which we refer to as *Client appends with no delegation*, or ***NoDelegation*** for short, the only way a client can append its record, is by issuing append operations directly to the corresponding DLOs, i.e., no other entity, including the other client, can do so. The second one, referred to as *Client appends with delegation*, or ***WithDelegation*** for short, is a relaxation of the first model, in which one client can append the record of the other client (if it knows it). Finally, in the third model, a record can be appended by an external (w.r.t. the clients) entity, provided it knows the record.

## 2.5 Client Model and Utility-based Problem Definitions

### 2.5.1 Client Setting

We assume that clients are ***rational***, i.e., they act selfishly, in a game-theoretic sense, in order to increase their utility [22]. Furthermore, clients are ***risk-averse***, i.e., when uncertain, they prefer to lower the uncertainty, even if this might lower their potential utility [22]; we consider a client to be uncertain when her actions may lead to multiple possible outcomes. To this respect, a rational, risk-averse client runs its own utility-driven protocol that defines its strategy towards a given protocol (game), in such a way that it would not decrease its utility or increase its uncertainty.

Regarding failures, the only type of failure we consider in this work, is ***crash failure***, in which a client might cease operating without any a priori warning.

Under this client model, ***an algorithm $\mathcal{A}$ solves the AtomicAppends problem***, if it provides enough incentive to the clients to follow this algorithm (which guarantees the safety and liveness properties of Definition 4, possibly in the presence of crashes), without any client deviating from its utility-driven protocol. If no such algorithm can be designed, then ***the AtomicAppends problem cannot be solved***.

### 2.5.2 Utility Models

Looking at the definition of the AtomicAppends problem, one might wonder what is the incentive of the clients to achieve this both-or-none principle on the appends. Let $U_X$ denote the utility function (or incentive) for each client $X$. A selfish rational client $X$ will try to maximize her utility $U_X$. Depending on the possible combinations of values the clients' utility functions can take, we can identify a number of different scenarios, we refer as *utility models*. Let us now motivate and specify two such utility models.

**Collaborative utility model.** Consider two clients $A$ and $B$ that have agreed to acquire a property (e.g., a piece of land) in common, and each has to provide half of the cost. If one of them, say $A$, pays while $B$ backs off from the deal, then $A$ incurs in expenses while not getting the property. On the other hand, $B$ loses no money in this case, but her reputation may suffer. If both of them back off, they do not have any cost, while if both proceed with the payments then they get the property, which they prefer.

If $U_X()$ denotes the utility of agent $X \in \{A, B\}$, then we have the following relations in the scenario described:

$$U_X(\text{both agents pay}) > U_X(\text{no agent pays}) > U_X(\text{only agent } \bar{X} \text{ pays}) > U_X(\text{only agent } X \text{ pays}).$$

| **Utility model** | **Utility of client X** |
|---|---|
| Collaborative | $U_X$(both append) $>$ $U_X$(none appends) $>$ $U_X$(only $\bar{X}$ appends) $>$ $U_X$(only $X$ appends) |
| Competitive | $U_X$(only $\bar{X}$ appends) $>$ $U_X$(both append) $>$ $U_X$(none appends) $>$ $U_X$(only $X$ appends) |

**Table 1** The utility of client $X \in \{A, B\}$ in the two utility models considered.

In relation to the AtomicAppends problem, record $r_A$ contains the transaction by which client $A$ pays her share of the deal, and the append of $r_A$ in DLO$_A$ carries out this payment. Similarly for client $B$. So, here we see that under the above utility model, both clients have incentive for both appends to take place. Observe that this situation is similar to the *Coordinated Attack* problem [10], in which two armies need to agree on attacking a common enemy. If both attack, then they win; if only one of them attacks, then that army is destroyed, while the other is disgraced; if none of them attack, then the status quo is preserved.

These utility examples fall in the general utility model depicted in the first row of Table 1, which we call *collaborative*. We will be referring to the AtomicAppends problem under this utility model as the ***Collaborative AtomicAppends*** problem.

**Competitive utility model.** We now consider a different utility model. Consider two clients $A$ and $B$ that have agreed to exchange their goods. E.g, $A$ gives his car to $B$, and $B$ gives a specific amount as payment to $A$. If one of them, say $A$, gives the car to $B$, but $B$ does not pay, then $A$ loses the car while not getting any money. On the other hand, $B$ gets the car for free! If both of them back off from the deal, then they do not have any cost. Both proceeding with the exchange is not necessarily their highest preference (unlike in the previous collaborative model).

So, if $U_X()$ denotes the utility of agent $X \in \{A, B\}$, then we have the following relations in the scenario described:

$$U_X(\text{only } \bar{X} \text{ proceeds}) > U_X(\text{both agents proceed}) > U_X(\text{no agent proceeds}) > U_X(\text{only } X \text{ proc.}).$$

In relation to the AtomicAppends problem, record $r_A$ contains the transaction transferring the deed of $A$'s car to $B$, and the append of $r_A$ in DLO$_A$ carries out this transfer. Similarly, $r_B$ contains the transaction by which client $B$ transfers a specific monetary amount to $A$ (pays for the car), and the append of $r_B$ in DLO$_B$ carries out this monetary transfer. Observe that this scenario is similar to the *Atomic Swaps* problem [13].

These utility examples fall in the general utility model depicted in the second row of Table 1, which we call *competitive*. We will be referring to the AtomicAppends problem under this utility model as the ***Competitive AtomicAppends*** problem.

No matter of the utility, failure or timing model assumed, our objective is to provide a solution to the AtomicAppends problem. Our investigation will focus on identifying the modeling conditions under which this is possible or not, and what is the impact of the model on the solvability of the problem.

## 3 AtomicAppends in the Absence of Client Crashes

We begin our investigation in a setting with no client crashes, so to study the impact of the utility model on the solvability of the problem.

It is not difficult to observe that in the absence of crash failures, even under asynchrony and NoDelegation, there is a straightforward algorithmic solution to the *Collaborative*

*AtomicAppends* problem: the algorithm simply has client $A$ (resp. client $B$) issuing operation *append*$(DLO_A, r_A)$ (resp. *append*$(DLO_B, r_B)$). Based on Table 1, the clients' utilities are maximized when both append their corresponding records. Since there are no failures and the DLOs are reliable, these operation are guaranteed to complete, nullifying the clients' uncertainty. Hence, the clients will follow the algorithm, without deviating from their utility-driven protocol. This yields the following result:

▶ **Theorem 5.** *Collaborative 2-AtomicAppends can be solved in the absence of failures, even under asynchrony and NoDelegation.*

However, this is *not* the case for the *Competitive AtomicAppends* problem. The problem cannot be solved, even in the absence of failures, in synchrony, and WithDelegation:

▶ **Theorem 6.** *Competitive 2-AtomicAppends cannot be solved in the absence of failures, even in synchrony and WithDelegation.*

**Proof.** Let us firstly show that client $A$ will never send its record $r_A$ to the other client $B$. The reason is that this would carry a large risk of $B$ appending $r_A$ itself (and $A$ is risk-averse). Observe that, independently on whether $B$ already appended $r_B$ or not, this would reduce $A$'s utility (see Table 1). Then, we secondly claim that client $A$ will not directly append its own record $r_A$ either. The reason is that, again, independently on whether $B$ already appended $r_B$ or not, this would reduce $A$'s utility (see Table 1). Hence, client $A$ will not have its record $r_A$ appended to $DLO_A$ ever. However, this violates the liveness property of Definition 4, since by assumption neither $A$ nor $B$ fail by crashing. ◀

Note that the above result does not contradict the known solutions for atomic swaps (e.g., [13]), as the primitives used are stronger than the ones offered by DLO (e.g., some form of validation is needed for hashlocks). As we show in Section 5, the problem can be solved in the model we consider, if a reliable external entity is used between the clients and the MDLO. In view of Theorems 5 and 6, in the next section we focus on the study of *Collaborative AtomicAppends* in the presence of crash failures.

## 4 Crash-prone Collaborative AtomicAppends with Client Appends

In this section we focus on the Collaborative AtomicAppend problem assuming that at least one client may crash, under the NoDelegation and WithDelegation client append models. Observe from Table 1 that both clients have incentive to get both records appended, versus the case of no record appended, with respect to utilities. However, as we will see, in some cases, crashes introduce uncertainty that renders the problem unsolvable.

### 4.1 Client Appends with No Delegation

We prove that *Collaborative AtomicAppends* cannot be guaranteed by any algorithm $\mathcal{A}$, even in a *synchronous system*, when at least one client crashes and the clients cannot delegate the append of their records.

▶ **Theorem 7.** *When at least one client crashes, Collaborative 2-AtomicAppends cannot be solved in the NoDelegation append model, even in a synchronous system.*

**Proof.** Consider an algorithm $\mathcal{A}$ that clients can execute without deviating from their utility-driven protocol. Assume algorithm $\mathcal{A}$ solves the Collaborative 2-AtomicAppends problem in the model described. Let $E$ be an execution of algorithm $\mathcal{A}$ in which no client crashes. By liveness, both clients $A$ and $B$ must issue append operations. Consider the first client, say $A$ without loss of generality, that issues the append operation. Let us assume that $A$ issues $\mathit{append}(\mathrm{DLO}_A, r_A)$ at time $t$. Hence, $B$ issues $\mathit{append}(DLO_B, r_B)$ at time no earlier than $t$, and $A$ cannot verify that the record $r_B$ is in the corresponding $DLO_B$ until time $t' > t$.

Now consider execution $E'$ of algorithm $\mathcal{A}$ that is identical to $E$, *up to time* $t$. Now at time $t$ client $B$ crashes, and hence it never issues $\mathit{append}(DLO_B, r_B)$. Since $A$ cannot differentiate until time $t$ this execution from $E$, it issues $\mathit{append}(DLO_A, r_A)$ at time $t$, appending $r_A$ to $DLO_A$. Even if after time $t$, $A$ detects the crash of client $B$, by the specification of NoDelegation, it cannot append record $r_B$ in $\mathrm{DLO}_B$. This, together with the fact that $B$ has crashed, yields that record $r_B$ is never appended to $DLO_B$, violating safety. Hence, we reach a contradiction, and algorithm $\mathcal{A}$ does not solve the Collaborative 2-AtomicAppends problem. ◀

## 4.2 Client Appends With Delegation

Let us now consider the more relaxed client append model of WithDelegation. It is not difficult to see that in this model, the impossibility proof of Theorem 7 breaks. In fact, it is easy to design an algorithm that solves the collaborative AtomicAppends problem in a synchronous system, if at most one client crashes. In a nutshell, first both clients exchange their records. When a client has both records, it appends them (one after the other) to the corresponding DLO; otherwise it does not append any record. We refer to this algorithm as Algorithm $\mathcal{A}_{DSync}$ and its pseudocode is given as Code 1. We show:

▶ **Theorem 8.** *In the WithDelegation append model, Algorithm* $\mathcal{A}_{DSync}$ *solves the Collaborative 2-AtomicAppends problem in a synchronous system, if at most one client crashes.*

**Proof.** If no client crashes, then the proof of the claim is straightforward. Hence, let us consider the case that one client crashes, say $A$. There are three cases:

(a) Client $A$ crashes before sending its record. In this case, client $B$ will not append any record and the problem is solved (none case).
(b) Client $A$ crashes after sending its record, but before it does any append. In this case client $B$ will receive $A$'s record and append both records (both case).
(c) Client $A$ crashes after it performs one or two of the appends. Client $B$ will perform both appends, and since DLOs guarantee that a record is appended only once (they are idempotent), the problem is solved (both case).

The above cases and Table 1 suggest that the clients have no risk in running Algorithm $\mathcal{A}_{DSync}$ with respect to their utility-driven protocol. Hence, the claim follows. ◀

We note that algorithm $\mathcal{A}_{DSync}$ solves the problem also in the asynchronous setting, without of course being able to implement the "else" statement (line 5), since in asynchrony, a client cannot distinguish the case on whether the other client has crashed or its message is taking too long to arrive. To this respect, we slightly modify the description of the algorithm to better highlight the inability to detect crashes. We refer to this version of the algorithm as $\mathcal{A}_{DAsync}$; its pseudocode is given as Code 2. We show:

▶ **Theorem 9.** *In the WithDelegation append model, Algorithm* $\mathcal{A}_{DAsync}$ *solves the Collaborative 2-AtomicAppends problem in an asynchronous system, if at most one client crashes.*

**Proof.** As before, we will prove this by case analysis. If no client crashes, then the proof follows easily, given the fact that a DLOs guarantees that a record is appended only once. Hence, let us consider the case that one client crashes, say $A$. There are three cases:

**Code 1** $\mathcal{A}_{DSync}$: AtomicAppends WithDelegation, Synchrony, at most one crash; code for Client $X \in \{A, B\}$.

```
send $r_X$ to client $\bar{X}$
If $r_{\bar{X}}$ is received from client $\bar{X}$ then
    append($DLO_X, r_X$)
    append($DLO_{\bar{X}}, r_{\bar{X}}$)
Else (client $\bar{X}$ has crashed)
    no append
```

**Code 2** $\mathcal{A}_{DAsync}$: AtomicAppends WithDelegation, Asynchrony, at most one crash; code for Client $X \in \{A, B\}$.

```
send $r_X$ to client $\bar{X}$
wait until $r_{\bar{X}}$ is received from client $\bar{X}$
    append($DLO_X, r_X$)
    append($DLO_{\bar{X}}, r_{\bar{X}}$)
```

(a) Client $A$ crashes before sending its record. In this case, client $B$ will not proceed to append any record (none case). Observe that client $B$ might not terminate, but the problem (safety) is not violated.
(b) Client $A$ crashes after sending its record, but before it does any append. In this case client $B$ will receive $A$'s record and append both records (both case).
(c) Client $A$ crashes after it performs one or two of the appends (it means it has sent its record to client $B$). Client $B$ will perform both appends, and since DLOs guarantee that a record is appended only once, the problem is solved (both case).

The above cases and Table 1 suggest that the clients have no risk in running Algorithm $\mathcal{A}_{DAsync}$ with respect to their utility-driven protocol. Hence, the claim follows. ◀

As already discussed in case (a) of the above proof, it is possible for the client that has not crashed to wait forever, as it cannot distinguish the case when the message is taking too long to arrive and the append operation is taking too long to complete, from the case when the other client has crashed. Hence, algorithm $\mathcal{A}_{DAsync}$, under certain conditions, is *non-terminating*[1].

Furthermore, it is not difficult to see that if both clients fail, neither algorithm $\mathcal{A}_{DAsync}$ nor algorithm $\mathcal{A}_{DSync}$ can solve the Collaborative AtomicAppends problem. For example, in the proof of Theorem 8, in case (b), client $B$ could crash right after appending its own record (i.e., $r_B$ is appended, but $r_A$ is not). This violates safety. In fact, we now show that if both clients can crash, the problem is not solvable, even under synchrony.

▶ **Theorem 10.** *When both clients can crash, the Collaborative 2-AtomicAppends problem cannot be solved WithDelegation, even in a synchronous system.*

**Proof.** Consider an algorithm $\mathcal{A}$ that clients can execute without deviating from their utility-driven protocol. Assume algorithm $\mathcal{A}$ solves the Collaborative 2-AtomicAppends problem in the model described. Let $E$ be an execution of algorithm $\mathcal{A}$ in which no client crashes. By liveness, both records $r_A$ and $r_B$ must be eventually appended. Consider the first record appended, say $r_A$ w.l.o.g., and the client that issued the append operation, say $A$ w.l.o.g.. Let

---

[1] Hence, in practice this may force a client to use timeouts in order to avoid blocking forever.

us assume that $A$ issues *append*(DLO$_A, r_A$) at time $t$. Hence, *append*($DLO_B, r_B$) is issued at time no earlier than $t$, and $A$ cannot verify that the record $r_B$ is in the corresponding $DLO_B$ until time $t' > t$.

Now consider execution $E'$ of algorithm $\mathcal{A}$ that is identical to $E$, *up to time* $t$. Now at time $t$ client $B$ crashes, and hence it never issues *append*($DLO_B, r_B$). Since $A$ cannot differentiate until time $t$ this execution from $E$, it issues *append*($DLO_A, r_A$) at time $t$, appending $r_A$ to $DLO_A$. Then, at time $t+1$ (immediately after *append*($DLO_A, r_A$) completes) $A$ also crashes, and hence never issues *append*($DLO_B, r_B$). Since *append*($DLO_B, r_B$) is never issued, record $r_B$ is never appended to $DLO_B$, violating safety. Hence, we reach a contradiction, and algorithm $\mathcal{A}$ does not solve the Collaborative 2-AtomicAppends problem. ◀

## 5 Crash-prone AtomicAppends with SDLO

Theorems 6 and 10 suggest the need to use some external intermediary entity, in order to solve *Competitive AtomicAppends*, even in the absence of crashes, and *Collaborative AtomicAppends*, in the case both clients crash, respectively. This is the subject of this section.

### 5.1 Smart DLO (SDLO)

We enhance the MDLO with a special DLO, called *Smart DLO* (SDLO), which is used by the clients to delegate the append of their records to the original MDLO. This SDLO is an extension of a DLO that supports a special "atomic appends" record of the form [**client id, {list of involved clients in the atomic append}, record of client**]. When two clients wish to perform an atomic append involving their records and their corresponding DLOs, then they *both* need to append such an atomic appends record in the SDLO; this is like requesting the atomic append service from the SDLO. Once *both* records are appended in the SDLO, then the SDLO appends each record to the corresponding DLO. A pseudocode of this mechanism, together with the client requests, called algorithm $\mathcal{A}_{SDLO}$ is given as Code 3.

**Code 3** $\mathcal{A}_{SDLO}$: SDLO mechanism and requests from client $X \in \{A, B\}$; SDLO code only for atomic appends

```
Client $X$:
    append($SDLO, [X, \{X, \bar{X}\}, r_X]$)
    upon receipt AppendAck from SDLO return
SDLO:
    Init: $S \leftarrow \emptyset$
    function SDLO.append($[X, \{X, \bar{X}\}, r_X]$)
        $S \leftarrow S \parallel [X, \{X, \bar{X}\}, r_X]$
        if $[\bar{X}, \{X, \bar{X}\}, r_{\bar{X}}] \in S$ then
            append($DLO_X, r_X$)
            append($DLO_{\bar{X}}, r_{\bar{X}}$)
        return AppendAck
```

So essentially the SDLO.`append` function in Code 3 can be viewed as a smart contract that "collects" the append requests involved in the AtomicAppends instance and ultimately executes them, by performing individual appends to the corresponding DLOs. Observe that the SDLO does not access the state of DLO$_A$ and DLO$_B$, but it needs to be able to perform append operations to both of them. In other words, delegation is passed to the SDLO. Also observe that the SDLO returns ack to a client's request, once their atomic appends request is appended in the SDLO, and not when the actual atomic append takes place.

## 5.2 Solving AtomicAppends with SDLO

It is not difficult to observe that algorithm $\mathcal{A}_{SDLO}$ can solve the AtomicAppends problem in both utility models, even *in asynchrony*, and even if *both clients crash.* Note that *SDLO*, being a distributed ledger by itself, is reliable despite the fact that some servers implementing it may fail (more below). We show:

▶ **Theorem 11.** *Algorithm $\mathcal{A}_{SDLO}$ solves both the Collaborative and Competitive 2-AtomicAppends problems in an asynchronous setting, even if both clients may crash.*

**Proof.** We consider three cases:

1. If no client crashes, then algorithm $\mathcal{A}_{SDLO}$ trivially solves the problem: Both clients invoke the atomic appends request to the SDLO, these operations complete, and the SDLO eventually triggers the two corresponding appends of records $r_A$ and $r_B$ to $DLO_A$ and $DLO_B$, respectively (both case).
2. At most one client crashes, say client $A$. Here we have two cases:
    a. Record $[A, \{A, B\}, r_A]$ is never appended to the SDLO. Since the SDLO will never contain both matching records, it will never append any of the records $r_A$ and $r_B$ (none case).
    b. Record $[A, \{A, B\}, r_A]$ is appended to the SDLO. Since record $[B, \{A, B\}, r_B]$ will eventually be appended by $B$ in the SDLO, it will proceed with the corresponding appends of records $r_A$ and $r_B$ (both case).
3. Both clients crash. If one of the two clients, say $A$, crashes before appending $[A, \{A, B\}, r_A]$ to the SDLO, then none of the appends of records $r_A$ and $r_B$ will take place in the corresponding DLOs (none case). However, if both clients crash after they have appended the matching atomic appends records, then both records $r_A$ and $r_B$ will be appended by the SDLO (both case).

Observe that the above hold for both utility models. In Competitive AtomicAppends, if a client does not invoke its atomic append request to the SDLO, it knows that the SDLO will not proceed to append the other client's record. This leaves the clients with their second best utility (see Table 1), and hence, both have incentive to invoke the atomic append requests to the SDLO. The reliability of the SDLO nullifies the uncertainty of the clients, and hence they will follow algorithm $\mathcal{A}_{SDLO}$. ◀

Observe that algorithm $\mathcal{A}_{SDLO}$ can easily be extended to solve the *k-AtomicAppend* problem, for any $k \geq 2$, provided that the utility of all records being appended is higher than none being appended for all clients: All clients submit their atomic append request to the SDLO, and then the SDLO performs the corresponding appends. Hence:

▶ **Corollary 12.** *Both the Collaborative and Competitive $k$-AtomicAppends problems can be solved with the use of SDLO in the asynchronous setting, even if all $k$ clients may crash.*

***Remark:*** As we discussed in the case 2 of the proof of Theorem 11, if client $A$ crashes and record $[A, \{A, B\}, r_A]$ is never appended to the SDLO, none of the records $r_A$ and $r_B$ will be appended. Now, observe that client $B$ can proceed to perform other operations once it has appended $[B, \{A, B\}, r_B]$ (despite the fact that $r_B$ has not been appended to DLO$_B$, as it is up to the SDLO to do so). Since clients do not need to wait forever for any operation, algorithm $\mathcal{A}_{SDLO}$ is terminating with respect to the clients. Moreover, the SDLO also terminates the processing of all the operations, as long as the appends in other DLOs terminate.

**Implementation issues.** In the above mechanism and theorem, we treat the SDLO as one entity. Since, however, the SDLO is a distributed ledger implemented by collaborating servers, there are some low-level implementation details that need to be discussed. If we assume that the servers implementing the SDLO are prone to only crash faults and that the SDLO is implemented using an Atomic Broadcast service, as described in [7], then algorithm $\mathcal{A}_{SDLO}$ can be implemented as follows: Clients $A$ and $B$ submit the atomic append requests to all servers implementing the SDLO. Once a server appends an atomic append request record to its local copy of the ledger, it checks if the matching record is already in the ledger. If this is the case, it issues the two corresponding append operations for records $r_A$ and $r_B$. If up to $f$ servers may crash, then it suffices that $f + 1$ servers, in total, perform these append operations. Given that each record is appended to a DLO at most once (the append operations are idempotent; if a record is already appended, it will not be appended again), it follows that both records are appended in the corresponding DLOs.

## 6 Conclusion

We have introduced the AtomicAppends problem, where given two (or more in general) clients, each needs to append a record to a corresponding DLO, and do so atomically with respect to each other: either both records are appended or none. We have considered crash-prone, rational and risk-averse clients based on two different utility models, *Collaborative* and *Competitive*, and studied the solvability of the problem under synchrony/asynchrony, different client append models and failure scenarios. Table 2 gives an overview of our results (for two clients): if the problem can be solved, then we list the algorithm we developed, otherwise we use the symbol "✘".

| | | **Synchrony** | | | **Asynchrony** | | |
|---|---|---|---|---|---|---|---|
| | | ND | WD | SDLO | ND | WD | SDLO |
| **Collaborative** | *no crashes* | simple | $\mathcal{A}_{DSync}$ | $\mathcal{A}_{SDLO}$ | simple | $\mathcal{A}^{(\star)}_{DAsync}$ | $\mathcal{A}_{SDLO}$ |
| | *up to one* | ✘ | | | ✘ | | |
| | *both* | | ✘ | | | ✘ | |
| **Competitive** | *no crashes* | ✘ | | | ✘ | | |
| | *up to one* | | | | | | |
| | *both* | | | | | | |

$^{(\star)}$ might not terminate

**Table 2** Overview of the results. ND stands for NoDelegation and WD for WithDelegation.

Our results demonstrate a clear separation on the solvability of the problem based on the utility model assumed when appends are done directly by the clients. When appends are done using a special type of a DLO, which we call *Smart* DLO (SDLO), then the problem is solved in both utility models, even in asynchrony and even if both clients may crash.

Our investigation of AtomicAppends did not look into the semantics of the records being appended. Consider, for example, the following scenario. Say that clients $A$ and $B$ initiate an atomic append request with records $r_A$ and $r_B$, respectively. While the atomic append request is being processed, say by the SDLO, client $B$ appends a record $r'$ directly to $DLO_B$. It could be the case that the content of record $r'$ is such, that it would affect record $r_B$. For example, say that the atomic append involves the exchange of a deed of a car with bitcoins; record $r_A$ contains the transfer of the deed and $r_B$ the transfer of bitcoins. If $r'$ involves the withdrawal of bitcoins from the wallet of client $B$, and this is appended first, then it could be the case that the wallet no longer contains sufficient bitcoins to carry out the atomic appends request. Even if we enforce the clients to perform all appends – not only atomic appends – through the SDLO (which practically speaking is not desirable), still we need to *validate* records. Therefore, to tackle such cases, we will need to consider *validated* DLOs (VDLOs) [7]. This is a challenging problem, especially in asynchronous settings.

## References

**1** Atomic swap. `https://en.bitcoin.it/wiki/Atomic_cross-chain_trading`.

**2** Cosmos. `https://cosmos.network`.

**3** Oraclize. `http://www.oraclize.it`.

**4** PolkaDot. `https://polkadot.network`.

**5** Elli Androulaki *et al.* Hyperledger fabric: a distributed operating system for permissioned blockchains. In *EuroSys 2018*, pp. 30:1–30:15.

**6** Elli Androulaki, Christian Cachin, Angelo De Caro, and Eleftherios Kokoris-Kogias. Channels: Horizontal scaling and confidentiality on permissioned blockchains. In *ESORICS 2018,* pp. 111-131.

**7** Antonio Fernandez Anta, Chryssis Georgiou, Kishori Konwar, and Nicolas Nicolaou. Formalizing and implementing distributed ledger objects. In *NETYS 2018.* Also, in SIGACT News, 49(2):58-76, 2018.

**8** Matthew K. Franklin and Gene Tsudik. Secure group barter: Multi-party fair exchange with semi-trusted neutral parties. In *FC 1998,* pp. 90–102.

**9** Mimi Gentz and Johnny Dude. Tunable data consistency levels in Microsoft Azure Cosmos DB. `https://docs.microsoft.com/en-us/azure/cosmos-db/consistency-levels`, 2017.

**10** Piotr J. Gmytrasiewicz and Edmund H. Durfee. Decision-theoretic recursive modeling and the coordinated attack problem. In *Proc.of 1992 Conference on AI Planning Systems*, pp. 88–95.

**11** Interledger W3C Community Group. Interledger. `https://interledger.org/`.

**12** Thomas Hardjono, Alexander Lipton, and Alex Pentland. Towards a design philosophy for interoperable blockchain systems. *CoRR*, abs/1805.05934, 2018.

**13** Maurice Herlihy. Atomic cross-chain swaps. In *PODC 2018,* pp. 245–254.

**14** Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM TOPLAS*, 12(3):463–492, 1990.

**15** Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690—691, 1979.

**16** N.A. Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers, 1996.

**17** Ron van der Meyden. On the specification and verification of atomic swap smart contracts. *CoRR*, abs/1811.06099, 2018.

**18** Silvio Micali, Michael O. Rabin, and Joe Kilian. Zero-knowledge sets. In *FOCS 2003,* pp. 80–91.

**19** Andrew Miller, Iddo Bentov, Ranjit Kumaresan, Christopher Cordi, and Patrick McCorry. Sprites and state channels: Payment networks that go faster than lightning. In *arXiv:1702.05812*, 2017.

**20** Aybek Mukhamedov, Steve Kremer, and Eike Ritter. Analysis of a multi-party fair exchange protocol and formal proof of correctness in the strand space model. In *FC 2005,* pp. 255–269.

**21** Arvind Narayanan, Joseph Bonneau, Edward W. Felten, Andrew Miller, and Steven Goldfeder. *Bitcoin and Cryptocurrency Technologies - A Comprehensive Introduction.* Princeton University Press, 2016.

**22** Martin J Osborne et al. *An introduction to Game Theory.* Oxford university press, 2004.

**23** Joseph Poon and Thaddeus Dryja. The bitcoin lightning network: Scalable off-chain instant payments. `https://lightning.network/lightning-network-paper.pdf`, 2016.

**24** Michel Raynal. *Concurrent Programming: Algorithms, Principles, and Foundations.* Springer, 2013.

**25** Michel Raynal. *Distributed Algorithms for Message-Passing Systems.* Springer, 2013.

**26** Matteo Gianpietro Zago. 50+ Examples of How Blockchains are Taking Over the World. *Medium*, 2018.