Formalizing and Implementing
Distributed Ledger Objects[*]

Antonio Fernández Anta[1], Chryssis Georgiou[2], Kishori Konwar[3], and
Nicolas Nicolaou[4]

[1] IMDEA Networks Institute, Madrid, Spain, `antonio.fernandez@imdea.org`
[2] University of Cyprus, Department of Computer Science, Nicosia, Cyprus,
`chryssis@cs.ucy.ac.cy`
[3] MIT, Cambridge, USA, `kishori@mit.edu`
[4] KIOS Research Center of Excellence, University of Cyprus, Nicosia, Cyprus
`nicolasn@ucy.ac.cy`

Abstract. Despite the hype about blockchains and distributed ledgers, no formal abstraction of these objects has been proposed[1]. To face this issue, in this paper we provide a proper formulation of a *distributed ledger object*. In brief, we define a *ledger* object as a sequence of *records*, and we provide the operations and the properties that such an object should support. Implementation of a ledger object on top of multiple (possibly geographically dispersed) computing devices gives rise to the *distributed ledger object*. In contrast to the centralized object, distribution allows operations to be applied concurrently on the ledger, introducing challenges on the *consistency* of the ledger in each participant. We provide the definitions of three well known consistency guarantees in terms of the operations supported by the ledger object: (1) *atomic consistency (linearizability)*, (2) *sequential consistency*, and (3) *eventual consistency*. We then provide implementations of distributed ledgers on asynchronous message passing crash-prone systems using an Atomic Broadcast service, and show that they provide eventual, sequential or atomic consistency semantics. We conclude with a variation of the ledger – the *validated ledger* – which requires that each record in the ledger satisfies a particular *validation rule*.

1   Introduction

We are living a huge hype of the so-called crypto-currrencies, and their technological support, the blockchain [21]. It is claimed that using crypto-currencies and public distributed ledgers (i.e., public blockchains) will liberate stakeholder owners from centralized trusted authorities [23]. Moreover, it is believed that there is the opportunity of becoming rich by mining coins, speculating with them, or even launching your own coin (i.e. with an initial coin offering, ICO).

---

[1] This observation was also pointed out by Maurice Herlihy in his PODC2017 keynote talk.

Cryptocurrencies were first introduced in 2009 by Satoshi Nakamoto [21]. In his paper, Nakamoto introduced the first algorithm that allowed economic transactions to be accomplished between peers without the need of a central authority. An initial analysis of the security of the protocol was presented in [21], although a more formal and thorough analysis was developed by Garay, Kiayias, and Leonardos in [10]. In that paper the authors define and prove two fundamental properties of the blockchain implementation behind bitcoin: (i) *common-prefix*, and (ii) *quality of chain*.

Although the recent popularity of distributed ledger technology (DLT), or blockchain, is primarily due to the explosive growth of numerous crypocurrencies, 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, a robustness and available system, and a system that provides security and privacy. For example, one rapidly emerging area of application of distributed ledger technology is medical and health care. At a high level, the distributed ledger can be used as a platform to store health care data for sharing, recording, analysis, research, etc. One of the most widely discussed approaches in adopting DLT is to implement Health Information Exchange (HIE), or share transactions among the participants such as patients, caregivers and other relevant parties [17]. Another interesting open-source initiative is Namecoin that uses DLT to improve the registration and ownership transfer of Internet components such as DNS [2]. Recently, in the real estate space there has been some experimental study to use DLT to implement a transparent and decentralized public ledger for managing land information, where the land registry serves property rights and historical transactions. Moreover, such an application would benefit from time stamping of transactions (a substitute for notarization); fault-tolerance from individual component crash (the system does not rely on a single centralized system); and non temper-able ledger of historical transactions and other blockchain applications to manage registry details [26]. Another example is to apply DLT in the management of scientific research projects to track and manage information such as publications, funding, and analysis in a publicly available, in a reproducible and transparent manner [1].

In the light of these works indeed crypto-currencies and (public and private) distributed ledgers[2] have the potential to impact our society deeply. However most experts, often do not clearly differentiate between the coin, the ledger that supports it, and the service they provide. Instead, they get very technical, talking about the cryptography involved, the mining used to maintain the ledger, or the smart contract technology used. Moreover, when asked for details it is often the case that there is no formal specification of the protocols, algorithms, and service provided, with a few exceptions [27]. In many cases "the code is the spec."

From the theoretical point of view there are many fundamental questions with the current distributed ledger (and crypto-currency) systems that are very often not properly answered: What is the service that must be provided by a distributed ledger? What properties a distributed ledger must satisfy? What are the assumptions made by the protocols and algorithms on the underlying system? Does a distributed ledger require a linked crypto-currency? In his PODC'2017 keynote address, Maurice Herlihy pointed

---

[2] We will use distributed ledger from now on, instead of blockchain.

Code 1 Ledger Object $\mathcal{L}$

```
1:  Init: S ← ∅
2:  function L.get( )
3:      return S
4:  function L.append(r)
5:      S ← S‖r
6:      return
```

Code 2 Validated Ledger Object $\mathcal{VL}$ (only append)

```
1:  function VL.append(r)
2:      if Valid(S‖r) then
3:          S ← S‖r
4:          return ACK
5:      else return NACK
```

out that, despite the hype about blockchains and distributed ledgers, no formal abstraction of these objects has been proposed [15]. He stated that there is a need for the formalization of the distributed systems that are at the heart of most cryptocurrency implementations, and leverage the decades of experience in the distributed computing community in formal specification when designing and proving various properties of such systems. In particular, he noted that the distributed ledger can be formally described by its sequential specification. Then, by using well-known concurrent objects, like consensus objects, come up with a universal construction of linearizable distributed ledgers.

In this paper we provide a proper formulation of a family of *ledger objects*, starting from a centralized, non replicated ledger object, and moving to distributed, concurrent implementations of ledger objects, subject to validation rules. In particular, we provide definitions and sample implementations for the following types of ledger objects:

- **Ledger Object (LO):** We begin with a formal definition of a *ledger object* as a sequence of *records*, supporting two basic operations: get and append. In brief, the ledger object is captured by Code 1 (in which ‖ is the concatenation operator), where the get operation returns the ledger as a sequence S of records, and the append operation inserts a new record at the end of the sequence. The *sequential specification* of the object is then presented, to explicitly define the expected behavior of the object when accessed sequentially by get and append operations.

- **Distributed Ledger Object (DLO):** With the ledger object implemented on top of multiple (possibly geographically dispersed) *computing devices* or *servers* we obtain *distributed ledgers* – the main focus of this paper. Distribution allows a (potentially very large) set of distributed *client processes* to access the distributed ledger, by issuing get and append operations concurrently. To explain the behavior of the operations during concurrency we define three consistency semantics: (i) eventual consistency, (ii) sequential consistency, and (iii) atomic consistency.

- **Implementations of DLO:** In light of our semantic definitions, we provide a number of algorithms that implement DLO that satisfy the above mentioned consistency semantics, on asynchronous message passing crash-prone systems, utilizing an Atomic Broadcast service.

- **Validated (Distributed) Ledger Object (V[D]LO):** We then provide a variation of the ledger object – the *validated ledger object* – which requires that each record in the ledger satisfies a particular *validation rule*, expressed as a predicate $Valid()$. To this end, the basic append operation of this type of ledger filters each record through the $Valid()$ predicate before is appended to the ledger (see Code 2).

Other related work. A distributed ledger can be used to implement a replicated state machine [18, 25]. Paxos [20] is one the first proposals of a replicated state machine

implemented with repeated consensus instances. The Practical Byzantine Fault Tolerance solution of Castro and Likov [7] is proposed to be used in Byzantine-tolerant blockchains. In fact, it is used by them to implement an asynchronous replicated state machine [6]. The recent work of Abraham and Malkhi [3] discusses in depth the relation between BFT protocols and blockchains consensus protocols. All these suggest that at the heart of implementing a distributed ledger object there is a version of a consensus mechanism, which directly impacts the efficiency of the implemented DLO. In a later section, we show that an eventual consistent DLO can be used to implement consensus, and consensus can be used to implement a DLO; this reinforces the relationship identified in the above-mentioned works.

Among the proposals for distributed ledgers, Algorand [12] is an algorithm for blockchain that boasts much higher throughput than Bitcoin and Ethereum. This work is a new resilient optimal Byzantine consensus algorithm targeting consortium blockchains. To this end, it first revisits the consensus validity property by requiring that the decided value satisfies a predefined predicate, which does not systematically exclude a value proposed only by Byzantine processes, thereby generalizing the validity properties found in the literature. In Gramoli's Red Belly system [13, 14], a blockchain is implemented on top of a modular Byzantine consensus algorithm that also relaxes the validity property of the commonly defined consensus problem.

One of the closest works to ours is the one by Anceaume et al [4], which like our work, attempts to connect the concept of distributed ledgers with distributed objects, although they concentrate in Bitcoin. In particular, they first show that read-write registers do not capture Bitcoin's behavior. To this end, they introduce the Distributed Ledger Register (DLR), a register that builds on read-write registers for mimicking the behavior of Bitcoin. In fact, they show the conditions under which the Bitcoin blockchain algorithm satisfies the DLR properties. Our work, although it shares the same spirit of formulating and connecting ledgers with concurrent objects (in the spirit of [22]), it differs in many aspects. For example, our formulation does not focus on a specific blockchain (such as Bitcoin), but aims to be more general, and beyond crypto-currencies. Hence, for example, instead of using sequences of blocks (as in [4]) we talk about sequences of records. Furthermore, following the concurrent object literature, we define the ledger object on new primitives (get and append), instead on building on multi-writer, multi-reader R/W register primitives. We pay particular attention on formulating the consistency semantics of the distributed ledger object and demonstrate their versatility by presenting implementations. Nevertheless, both works, although taking different approaches, contribute to the better understanding of the basic underlying principles of distributed ledgers from the theoretical distributed computing point of view.

2 The Ledger Object

In this section we provide the fundamental definition of a concurrent ledger object.

2.1 Concurrent Objects and the Ledger Object

An *object type* T specifies (i) the set of *values* (or states) that any object O of type T can take, and (ii) the set of *operations* that a process can use to modify or access the value

of O. An object O of type T is a *concurrent object* if it is a shared object accessed by multiple processes [24]. Each operation on an object O consists of an *invocation* event and a *response* event, that must occur in this order. A *history* of operations on O, denoted by H_O, is a sequence of invocation and response events, starting with an invocation event. (The sequence order of a history reflects the real time ordering of the events.) An operation π is *complete* in a history H_O, if H_O contains both the invocation and the matching response of π, in this order. A history H_O is *complete* if it contains only complete operations; otherwise it is *partial* [24]. An operation π_1 *precedes* an operation π_2 (or π_2 *succeeds* π_1), denoted by $\pi_1 \to \pi_2$, in H_O, if the response event of π_1 appears before the invocation event of π_2 in H_O. Two operations are *concurrent* if none precedes the other.

A complete history H_O is *sequential* if it contains no concurrent operations, i.e., it is an alternative sequence of matching invocation and response events, starting with an invocation and ending with a response event. A partial history is sequential, if removing its last event (that must be an invocation) makes it a complete sequential history. A *sequential specification* of an object O, describes the behavior of O when accessed sequentially. In particular, the sequential specification of O is the set of all possible sequential histories involving solely object O [24].

A *ledger* $\mathcal{L}$ is a concurrent object that stores a totally ordered sequence $\mathcal{L}.S$ of *records* and supports two operations (available to any process p): (i) $\mathcal{L}.\text{get}_p()$, and (ii) $\mathcal{L}.\text{append}_p(r)$. A *record* is a triple $r = \langle \tau, p, v \rangle$, where τ is a *unique* record identifier from a set $\mathcal{T}$, $p \in \mathcal{P}$ is the identifier of the process that created record r, and v is the data of the record drawn from an alphabet A. 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 a $\mathcal{L}.\text{get}_p()$ operation[3] to obtain the sequence $\mathcal{L}.S$ of records stored in the ledger object $\mathcal{L}$, and p invokes a $\mathcal{L}.\text{append}_p(r)$ operation to extend $\mathcal{L}.S$ with a new record r. Initially, the sequence $\mathcal{L}.S$ is empty.

Definition 1. *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 π in $H_{\mathcal{L}}$ the value of the sequence in ledger $\mathcal{L}$ is $\mathcal{L}.S = V$, then:*

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

2.2 Implementation of Ledgers

Processes execute operations and instructions sequentially (i.e., we make the usual well-formedess assumption that a process will not invoke an operation before a previously invoked one has not completed). A process p interacts with a ledger $\mathcal{L}$ by invoking an operation ($\mathcal{L}.\text{get}_p()$ or $\mathcal{L}.\text{append}_p(r)$), which causes a request to be sent to the ledger, and waiting for a response, which marks the end of the operation[4]. The response carries

---

[3] We define only one operation to access the value of the ledger for simplicity. In practice, other operations, like those to access individual records in the sequence, will also be available.

[4] We make explicit the exchange of request and responses between the process and the ledger to reveal the fact that the ledger is concurrent, i.e., accessed by several processes.
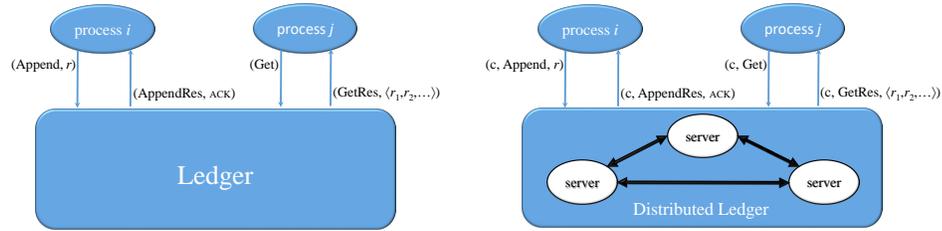
Fig. 1. The interaction between processes and the ledger, where $r, r_1, r_2, \ldots$ are records. **Left**: General abstraction; **Right**: Distributed ledger implemented by servers

Code 3 External Interface (Executed by a Process p) of a Ledger Object $\mathcal{L}$	**Code 4** Ledger $\mathcal{L}$ (centralized)

```
 1: function L.get( )
 2:     send request (GET) to ledger L
 3:     wait response (GETRES, V) from L
 4:     return V
 5: function L.append(r)
 6:     send request (APPEND, r) to ledger L
 7:     wait response (APPENDRES, res) from L
 8:     return res
```

```
 1: Init: S ← ∅
 2: receive (GET) from process p
 3:     send response (GETRES, S) to p
 4: receive (APPEND, r) from process p
 5:     S ← S‖r
 6:     send resp (APPENDRES, ACK) to p
```

the result of the operation. The result for a get operation is a sequence of records, while the result for an append operation is a confirmation (ACK). This interaction from the point of view of the process p is depicted in Code 3. A possible centralized implementation of the ledger that processes requests sequentially is presented in Code 4 (each block **receive** is assumed to be executed in mutual exclusion). Figure 1(left) abstracts the interaction between the processes and the ledger.

3 Distributed Ledger Objects

In this section we define distributed ledger objects, and some of the levels of consistency guarantees that they can provided. Then, we show how to implement distributed ledger objects that satisfy these consistency levels using an atomic broadcast [9] service.

3.1 Distributed Ledgers and Consistency

Distributed Ledgers A *distributed ledger object* (distributed ledger for short) is a concurrent ledger object that is implemented in a distributed manner. In particular, the ledger object is *implemented* by (and possibly replicated among) a set of (possibly distinct and geographically dispersed) computing devices, that we refer as *servers*. We refer to the processes that invoke the get() and append() operations of the distributed ledger as *clients*. Figure 1(right) depicts the interaction between the clients and the distributed ledger, implemented by servers.

In general, servers can fail. This leads to introducing mechanisms in the algorithm that implements the distributed ledger to achieve fault tolerance, like replicating the

ledger. Additionally, the interaction of the clients with the servers will have to take into account the faulty nature of individual servers, as we discuss later in the section.

Consistency of Distributed Ledgers Distribution and replication intend to ensure availability and survivability of the ledger, in case a subset of the servers fails. At the same time, they raise the challenge of maintaining *consistency* among the different views that different clients get of the distributed ledger: what is the latest value of the ledger when multiple clients may send operation requests at different servers concurrently? Consistency semantics need to be in place to precisely describe the allowed values that a get() operation may return when it is executed concurrently with other get() or append() operations. Here, as examples, we provide the properties that operations must satisfy in order to guarantee *atomic consistency* (linearizability) [16], *sequential consistency* [19] and *eventual consistency* [11] semantics. In a similar way, other consistency guarantees, such as session and causal consistencies could be formally defined [11].

Atomicity (aka, linearizability) [5, 16] provides the illusion that the distributed ledger is accessed sequentially respecting the real time order, even when operations are invoked concurrently. I.e., the distributed ledger seems to be a centralized ledger like the one implemented by Code 4. Formally[5],

Definition 2. *A distributed ledger $\mathcal{L}$ is* atomic *if, given any complete history $H_{\mathcal{L}}$, there exists a permutation σ of the operations in $H_{\mathcal{L}}$ such that:*

1. *σ follows the sequential specification of $\mathcal{L}$, and*
2. *for every pair of operations π_1, π_2, if $\pi_1 \rightarrow \pi_2$ in $H_{\mathcal{L}}$, then π_1 appears before π_2 in σ.*

Sequential consistency [5, 19] is weaker than atomicity in the sense that it only requires that operations respect the local ordering at each process, not the real time ordering. Formally,

Definition 3. *A distributed ledger $\mathcal{L}$ is* sequentially consistent *if, given any complete history $H_{\mathcal{L}}$, there exists a permutation σ of the operations in $H_{\mathcal{L}}$ such that:*

1. *σ follows the sequential specification of $\mathcal{L}$, and*
2. *for every pair of operations π_1, π_2 invoked by a process p, if $\pi_1 \rightarrow \pi_2$ in $H_{\mathcal{L}}$, then π_1 appears before π_2 in σ.*

Let us finally give a definition of eventually consistent distributed ledgers. Informally speaking, a distributed ledger is eventual consistent, if for every append(r) operation that completes, *eventually* all get() operations return sequences that contain record r, and in the same position. Formally,

Definition 4. *A distributed ledger $\mathcal{L}$ is* eventually consistent *if, given any complete history $H_{\mathcal{L}}$, there exists a permutation σ of the operations in $H_{\mathcal{L}}$ such that:*

1. *σ follows the sequential specification of $\mathcal{L}$, and*

---

[5] Our formal definitions of linearizability and sequential consistency are adapted from [5].

2. *for every $\mathcal{L}$.append$(r) \in H_\mathcal{L}$, there exists a complete history $H'_\mathcal{L}$ that extends[6] $H_\mathcal{L}$ such that, for every complete history $H''_\mathcal{L}$ that extends $H'_\mathcal{L}$, every complete operation $\mathcal{L}$.get() in $H''_\mathcal{L} \setminus H'_\mathcal{L}$ returns a sequence that contains r.*

Remark: Observe that in the above definitions we consider $H_\mathcal{L}$ to be complete. As argued in [24], the definitions can be extended to sequences that are not complete by reducing the problem of determining whether a complete sequence extracted by the non complete one is consistent. That is, given a partial history $H_\mathcal{L}$, if $H_\mathcal{L}$ can be modified in such a way that every invocation of a non complete operation is either removed or completed with a response event, and the resulting, complete, sequence $H'_\mathcal{L}$ checks for consistency, then $H_\mathcal{L}$ also checks for consistency. Alternatively, following [5], a liveness assumption can be made where every invocation event has a matching response event (and hence all histories are complete).

3.2 Distributed Ledger Implementations

In this section we provide implementations of distributed ledgers with different levels of consistency, as a mean of illustrating the generality and versatility of our ledger formulation. These implementations build on a generic deterministic atomic broadcast service [9].

Distributed Setting We consider an asynchronous message-passing distributed system. There is an unbounded number of clients accessing the distributed ledger. There is a set $\mathcal{S}$ of n servers, that emulate a ledger (c.f., Code 4) in a distributed manner. Both clients and servers might fail by crashing. However, no more than $f < n$ of servers might crash[7]. Processes (clients and servers) interact by message passing communication over asynchronous reliable channels.

We assume that clients are aware of the faulty nature of servers and know (an upper bound on) the maximum number of faulty servers f. Hence, we assume they use a modified version of the interface presented in Code 3 to deal with server unreliability. The new interface is presented in Code 5. As can

Code 5 External Interface of a Distributed Ledger Object $\mathcal{L}$ Executed by a Process p

```
1:  c ← 0
2:  Let L ⊆ S : |L| ≥ f + 1
3:  function L.get( )
4:      c ← c + 1
5:      send request (c, GET) to the servers in L
6:      wait response (c, GETRES, V) from some server in L
7:      return V
8:  function L.append(r)
9:      c ← c + 1
10:     send request (c, APPEND, r) to the servers in L
11:     wait response (c, APPENDRES, res) from some server in L
12:     return res
```

be seen there, every operation request is sent to a set L of at least $f + 1$ servers, to guarantee that at least one correct server receives and processes the request (if an upper bound on f is not known, then the clients contact all servers). Moreover, at least one

---

[6] A sequence X extends a sequence Y when Y is a prefix of X.

[7] The atomic broadcast service used in the algorithms may internally have more restrictive requirements.

such correct server will send a response which guarantees the termination of the operations. For formalization purposes, the first response received for an operation will be considered as the response event of the operation. In order to differentiate from different responses, all operations (and their requests and responses) are uniquely numbered with counter c, so duplicated responses will be identified and ignored (i.e., only the first one will be processed by the client).

In the remainder of the section we focus on providing the code run by the servers, i.e., the distributed ledger emulation. The servers will take into account Code 5, and in particular the fact that clients send the same request to multiple servers. This is important, for instance, to make sure that the same record r is not included in the sequence of records of the ledger multiple times. As already mentioned, our algorithms will use as a building block an atomic broadcast service. Consequently, our algorithms' correctness depends on the modeling assumptions of the specific atomic broadcast implementation used. We now give the guarantees that our atomic broadcast service need to provide.

Atomic Broadcast Service The Atomic Broadcast service (aka, total order broadcast service) [9] has two operations: ABroadcast(m) used by a server to broadcast a message m to all servers $s \in \mathcal{S}$, and ADeliver(m) used by the atomic broadcast service to deliver a message m to a server. The following properties are guaranteed (adopted from [9]):

- *Validity*: if a correct server broadcasts a message, then it will eventually deliver it.
- *Uniform Agreement*: if a server delivers a message, then all correct servers will eventually deliver that message.
- *Uniform Integrity*: a message is delivered by each server at most once, and only if it was previously broadcast.
- *Uniform Total Order*: the messages are totally ordered; that is, if any server delivers message m before message m', then every server that delivers them, must do it in that order.

Eventual Consistency and relation with Consensus We now use the Atomic Broadcast service to implement distributed ledgers in our set of servers $\mathcal{S}$ guaranteeing different consistency semantics. We start by showing that the algorithm presented in Code 6 implements an eventually consistent ledger, as specified in Definition 4.

Code 6 Eventually Consistent Distributed Ledger $\mathcal{L}$; Code for Server $i \in \mathcal{S}$

```
1: Init: S_i ← ∅
2: receive (c, GET) from process p
3:     send response (c, GETRES, S_i) to p
4: receive (c, APPEND, r) from process p
5:     ABroadcast(r)
6:     send response (c, APPENDRES, ACK) to p
7: upon (ADeliver(r)) do
8:     if r ∉ S_i then S_i ← S_i‖r
```

Code 7 Consensus Algorithm Using an Eventually Consistent Ledger $\mathcal{L}$

```
1: function propose(v)
2:     L.append(v)
3:     V_i ← L.get()
4:     while V_i = ∅ do
5:         V_i ← L.get()
6:     decide the first value in V_i
```

Lemma 1. *The combination of the algorithms presented in Code 5 and Code 6 implements an eventually consistent distributed ledger.*

Proof. The lemma follows from the properties of atomic broadcast. Considering any complete history $H_{\mathcal{L}}$, the permutation σ that follows the sequential specification can be defined as follows. The $\mathcal{L}.\mathsf{append}(r)$ operations are sorted in σ in the order the atomic broadcast service delivers the first copy of the corresponding records r (which is the same for all servers that receive them). (Observe that any later delivery of r is discarded.) Then, consider any $\pi = \mathcal{L}.\mathsf{get}()$ operation. Let V be the sequence returned by π and r the last record in V. Then, π is placed in σ after the operation $\mathcal{L}.\mathsf{append}(r)$ and before the next append operation. If V is empty, then π is placed before any append operation. The get operations are sorted randomly between append operations. The algorithm in Code 6 guarantees that this permutation σ follows the sequential specification.

Now, let us now consider a $\mathcal{L}.\mathsf{append}(r)$ operation in $H_{\mathcal{L}}$, which by definition is completed. From Code 5 at least one correct server i received the request $(c, append, r)$. Then, from Code 6, server i invoked $\mathsf{ABroadcast}(r)$. By validity, the record r is delivered to server i. Then, the uniform agreement and uniform total order properties of the atomic broadcast service guarantee that r will be delivered by all correct servers. The first time r is delivered to a server j, it includes it in the sequence S_j. Uniform total order guarantees that the first delivery of r happens in the same position of S_j in all servers j that deliver r. Any later delivery of r to server j is discarded (since r is already in the sequence) so r appears only once in the server sequences. Hence, after r is included in the sequences of all correct servers and all faulty servers have crashed, all get operations return sequences that contain r. □

Let us now explore the power of any eventually consistent distributed ledger. It is known that atomic broadcast is equivalent to consensus in a crash-prone system like the one considered here [8]. Then, the algorithm presented in Code 6 can be implemented as soon as a consensus object is available. What we show now is that a distributed ledger that provides the eventual consistency can be used to solve the consensus problem, defined as follows.

Consensus Problem: Consider a system with at least one non-faulty process and in which each process p_i proposes a value v_i from the set V (calling a $\mathsf{propose}(\mathsf{v_i})$ function), and then decides a value $o_i \in V$, called the *decision*. Any decision is irreversible, and the following conditions are satisfied: (i) *Agreement*: All decision values are identical. (ii) *Validity:* If all calls to the propose function that occur contain the same value v, then v is the only possible decision value. and (iii) *Termination:* In any fair execution every non-faulty process decides a value.

Lemma 2. *The algorithm presented in Code 7 solves the consensus problem if the ledger $\mathcal{L}$ guarantees eventual consistency.*

Proof. A correct process p that invokes $\mathsf{propose}_p(v)$ will complete its $\mathcal{L}.\mathsf{append}_p(v)$ operation. Hence, eventual consistency guarantees that it will terminate, since eventually some $\mathcal{L}.\mathsf{get}_p()$ will return a non-empty sequence. The agreement and validity properties of consensus are also satisfied since eventual consistency guarantees that all $\mathcal{L}.\mathsf{get}()$ operations that return non-empty sequences, return the same value in the first position of the sequence. □

Combining the above arguments and lemmas we have the following theorem.

Theorem 1. *Consensus and eventually consistent distributed ledgers are equivalent in a crash-prone distributed system.*

Atomic Consistency Observe that the eventual consistent implementation does not guarantee that record r has been added to the ledger before a response APPENDRES is received by the client p issuing the append(r). This may lead to situations in which a client may complete an append() operation, and a succeeding get() may not contain the appended record. This behavior is also apparent in Definition 4, that allows any get() operation, that is invoked and completed in $H'_{\mathcal{L}}$, to return a sequence that does not include a record r which was appended by an append(r) operation that appears in $H_{\mathcal{L}}$.

An *atomic distributed ledger* avoids this problem and requires that a record r appended by an append(r) operation, is received by any succeeding get() operation, even if the two operations where invoked at different processes. Code 8, describes the algorithm at the servers in order to implement an atomic consistent distributed ledger. The algorithm of each client is depicted from Code 5. Briefly, when a server receives a *get* or an *append* request, it adds the request in a pending set and atomically broadcasts the request to all other servers. When an append or get message is delivered, then the server replies to the requesting process (if it did not reply yet).

Code 8 Atomic Distributed Ledger; Code for Server i

```
 1: Init: S_i ← ∅; get_pending_i ← ∅; pending_i ← ∅
 2: receive (c, GET) from process p
 3:     ABroadcast(get, p, c)
 4:     add (p, c) to get_pending_i
 5: receive (c, APPEND, r) from process p
 6:     ABroadcast(append, r)
 7:     add (c, r) to pending_i
 8: upon (ADeliver(append, r)) do
 9:     if r ∉ S_i then
10:         S_i ← S_i‖r
11:         if ∃(c, r) ∈ pending_i then
12:             send response (c, APPENDRES, ACK) to r.p
13:             remove (c, r) from pending_i
14: upon (ADeliver(get, p, c)) do
15:         if (p, c) ∈ get_pending_i then
16:             send response (c, GETRES, S_i) to p
17:             remove (p, c) from get_pending_i
```

Theorem 2. *The combination of the algorithms presented in Codes 8 and 5 implements an atomic distributed ledger.*

Proof. To show that atomic consistency is preserved, we need to prove that our algorithm satisfies the properties presented in Definition 2. The underlying atomic broadcast defines the order of events when operations are concurrent. It remains to show that operations that are separate in time can be ordered with respect to their real time ordering. The following properties capture the necessary conditions that must be satisfied by non-concurrent operations that appear in a history $H_{\mathcal{L}}$:

A1 if $\mathsf{append}_{p_1}(r_1) \to \mathsf{append}_{p_2}(r_2)$ from processes p_1 and p_2, then r_1 must appear before r_2 in any sequence returned by the ledger

A2 if $\mathsf{append}_{p_1}(r_1) \to \mathsf{get}_{p_2}()$, then r_1 appears in the sequence returned by $\mathsf{get}_{p_2}()$

A3 if π_1 and π_2 are two get() operations from p_1 and p_2, s.t. $\pi_1 \to \pi_2$, that return sequences S_1 and S_2 respectively, then S_1 must be a prefix of S_2

A4 if $\mathsf{get}_{p_1}() \to \mathsf{append}_{p_2}(r_2)$, then p_1 returns a sequence S_1 that does not contain r_2

Property, **A1** is preserved from the fact that record r_1 is atomically broadcasted and delivered before r_2 is broadcasted among the servers. In particular, let p_1 be the process that invokes $\pi_1 = \mathsf{append}_{p_1}(r_1)$, and p_2 the process that invokes $\pi_2 = \mathsf{append}_{p_2}(r_2)$

(p_1 and p_2 may be the same process). Since $\pi_1 \rightarrow \pi_2$, then p_1 receives a response to the π_1 operation, before p_2 invokes the π_2 operation. Let server s be the first to respond to p_1 for π_1. Server s sends a response only if the procedure ADeliver$(append, r_1)$ occurs at s. This means that the atomic broadcast service delivers $(append, r_1)$ to s. Since $\pi_1 \rightarrow \pi_2$ then no server received the append request for π_2, and thus r_2 was not broadcasted before the ADeliver$(append, r_1)$ at s. Hence, by the *Uniform Total Order* of the atomic broadcast, every server delivers $(append, r_1)$ before delivering $(append, r_2)$. Thus, the ADeliver$(append, r_2)$ occurs in any server s' after the appearance of ADeliver$(append, r_1)$ at s'. Therefore, if s' is the first server to reply to p_2 for π_2, it must be the case that s' added r_1 in his ledger sequence before adding r_2.

In similar manner we can show that property **A2** is also satisfied. In particular let processes p_1 and p_2 (not necessarily different), invoke operations $\pi_1 = \mathsf{append}_{p_1}(r_1)$ and $\pi_2 = \mathsf{get}_{p_2}()$, s.t. $\pi_1 \rightarrow \pi_2$. Since π_1 completes before π_2 is invoked then there exists some server s in which ADeliver$(append, r_1)$ occurs before responding to p_1. Also, since the GET request from p_2 is sent, after π_1 has completed, then it follows that is sent after ADeliver$(append, r_1)$ occured in s. Therefore, (get, p_2, c) is broadcasted after ADeliver$(append, r_1)$ as well. Hence by *Uniform Total Order* atomic broadcast, every server delivers $(append, r_1)$ before delivering (get, p_2, c). So if s' is the first server to reply to p_2, it must be the case that s' received $(append, r_1)$ before receiving (get, p_2, c) and hence replies with an S_i to p_2 that contains r_1.

The proof of property **A3** is slightly different. Let $\pi_1 = \mathsf{get}_{p1}()$ and $\pi_2 = \mathsf{get}_{p2}()$, s.t. $\pi_1 \rightarrow \pi_2$. Since π_1 completes before π_2 is invoked then the (get, p_1, c_1) must be delivered to at least a server s that responds to p_1, before the invocation of π_2, and thus the broadcast of (get, p_2, c_2). By *Uniform Total Order* again, all servers deliver (get, p_1, c_1) before delivering (get, p_2, c_2). Let S_1 be the sequence sent by s to p_1. Notice that S_1 contains all the records r such that $(append, r)$ delivered to s before the delivery of (get, p_1, c_1) to s. Thus, for every r in S_1, ADeliver$(append, r)$ occurs in s before ADeliver(get, p_1, c). Let s' be the first server that responds for π_2. By *Uniform Agreement*, since s' has not crashed before responding to p_2, then every r in S_1 that was delivered in s, was also delivered in s'. Also, by *Uniform Total Order*, it must be the case that all records in S_1 will be delivered to s' in the same order that have been delivered to s. Furthermore all the records will be delivered to s' before the delivery of $(get, p_1, c1)$. Thus, all records are delivered at server s' before (get, p_2, c_2) as well, and hence the sequence S_2 sent by s' to p_2 is a suffix of S_1.

Finally, if $\mathsf{get}_{p_1}() \rightarrow \mathsf{append}_{p_2}(r_2)$ as in property **A4**, then trivially p_1 cannot return r_2, since it has not yet been broadcasted (*Uniform Integrity* of the atomic broadcast). □

Sequential Consistency An atomic distributed ledger also satisfies sequential consistency. As sequential consistency is weaker than atomic consistency, one may wonder whether a sequentially consistent ledger can be implemented in a simpler way.

We propose here an implementation, depicted in Code 9, that avoids the atomic broadcast of the get requests. Instead, it applies some changes to the client code to achieve sequential consistency, as presented in Code 10. This implementation provides both sequential (cf. Definition 3) and eventual consistency (cf. Definition 4).

Theorem 3. *The combination of the algorithms presented in Code 9 and Code 10 implements a sequentially consistent distributed ledger.*

Code 9 Sequentially Consistent Distributed Ledger; Code for Server i

```
1:  Init: S_i ← ∅; pending_i ← ∅; get_pending_i ←
    ∅
2:  receive (c, GET, ℓ) from process p
3:      if |S_i| ≥ ℓ then
4:          send response (c, GETRES, S_i) to p
5:      else
6:          add (c, p, ℓ) to get_pending_i
7:  receive (c, APPEND, r) from process p
8:      ABroadcast(c, r)
9:      add (c, r) to pending_i
10: upon (ADeliver(c, r)) do
11:     if r ∉ S_i then S_i ← S_i‖r
12:     if (c, r) ∈ pending_i then
13:         send resp. (c, APPENDRES, ACK, |S_i|) to r.p
14:         remove (c, r) from pending_i
15:     if ∃(c', p, ℓ) ∈ get_pending_i : |S_i| ≥ ℓ then
16:         send response (c', GETRES, S_i) to p
17:         remove (c', p, ℓ) from get_pending_i
```

Code 10 External Interface for Sequential Consistency Executed by a Process p

```
1:  c ← 0; ℓ_last ← 0
2:  Let L ⊆ S : |L| ≥ f + 1
3:  function L.get( )
4:      c ← c + 1
5:      send request (c, GET, ℓ_last) to the servers in L
6:      wait response (c, GETRES, V) from some server in
        L
7:      ℓ_last ← |V|
8:      return V
9:  function L.append(r)
10:     c ← c + 1
11:     send request (c, APPEND, r) to the servers in L
12:     wait response (c, APPENDRES, res, pos) from
        some server in L
13:     ℓ_last ← pos
14:     return res
```

Proof. Sequential consistency is preserved if we can show that in any history $H_{\mathcal{L}}$, the following properties are satisfied for any process p :

S1 if $\mathsf{append}_p(r_1) \to \mathsf{append}_p(r_2)$ then r_1 must appear before r_2 in the sequence of the ledger.

S2 if $\pi = \mathsf{get}_p() \to \mathsf{append}_p(r_1)$, then π returns a sequence V_p that does not contain r_1

S3 if $\mathsf{append}_p(r_1) \to \mathsf{get}_p() = \pi$, then π returns a sequence V_p that contains r_1

S4 if π_1 and π_2 are two $\mathsf{get}_p()$ operations, s.t. $\pi_1 \to \pi_2$, that return sequences V_1 and V_2 respectively, then V_1 must be a prefix of V_2.

Property **S1** is preserved as p waits for a response to the $\mathsf{append}_p(r_1)$ operation before invoking the $\mathsf{append}_p(r_2)$ operation. Let server s be the first to respond to the $\mathsf{append}_p(r_1)$. Server s sends a response only if (c_1, r_1) is delivered at s. Since r_2 was not broadcasted before the $\mathsf{ADeliver}(c_1, r_1)$ at s then by the *Uniform Total Order* of the atomic broadcast, every server will receive r_1 before receiving r_2. Thus, the $\mathsf{ADeliver}(c_2, r_2)$ occurs in a server s' after the appearance of $\mathsf{ADeliver}(c_1, r_1)$. Hence, if s' is the first server to reply to p for $\mathsf{append}_p(r_2)$, it must be the case that s' added r_1 in his ledger sequence before adding r_2.

Property **S2** is also preserved because p waits for $\mathsf{get}_p()$ to complete before invoking $\mathsf{append}_p(r_1)$. Since r_1 is broadcasted to the servers after $\mathsf{get}_p()$ is completed, then by *Uniform Integrity*, it is impossible that record r_1 was included in the sequence returned by the $\mathsf{get}_p()$ operation.

Property **S3** holds given that a server responds to the get operation of p only if its local sequence is longer than the position where the last record r of p was appended. In particular, from Code 10, the $\mathsf{append}_p(r)$ operation completes if a server s delivers the (c, r) message and responds to p. Within the response, s encloses the length ℓ of his local sequence once appended with r. When p invokes the $get_p()$ operation, it encloses ℓ within its GET request. Let s' be the first server that replies to the $get_p()$ with a sequence $V_{s'}$. From Code 9, it follows that s' responds to p only when $|V_{s'}| \geq \ell$. By the *Uniform*

Total Order it follows that all record messages are delivered in the same order in every server (including s'). Hence, if r was appended at location ℓ in s, then r appears in position ℓ in the sequence of s' as well. Since $|V_{s'}| \geq \ell$ then it must be the case that $V_{s'}$ contains r at position ℓ. So $get_p()$ returns a sequence that contains r as needed.

Similarly, Property **S4** holds as a get operation stores the length of the obtained sequence and the next get from the same process passes this length along with its GET request. With similar reasoning as in the proof of property **S3**, if π_1 and π_2 are two $get_p()$ operations from the same process p, s.t. $\pi_1 \rightarrow \pi_2$, and return V_1 and V_2 respectively, then it must be the case that $|V_1| \leq |V_2|$. From the *Uniform Total Order* the appends are received in the same order in every server, and hence V_1 is a prefix of V_2.

Hence, the only pending issue is to prove that the wait instruction in Lines 6-7 of Code 10 will eventually terminate. This means that some server in L must return a sequence V that is longer or equal to the latest sequence, say with length ℓ, received at p. There are two cases to consider: (i) either p received ℓ as a result of an append operation, or (ii) p received a sequence with length ℓ as a result of a get operation. Let server s be the one to respond to p with a length ℓ or with a sequence of length ℓ. It follows that ℓ records are delivered to s. By *Uniform Agreement*, these records will be delivered to all correct processes. Since there will be at least a single server s' to be correct at any $get_p()$ operation of p, then s' will eventually receive all the records and will respond to p. $\square$

4 Validated Ledgers

A *validated ledger* $\mathcal{VL}$ is a ledger in which specific semantics are imposed on the contents of the records stored in the ledger. For instance, if the records are (bitcoin-like) financial transactions, the semantics should, for example, prevent double spending, or apply other transaction validation used as part of the Bitcoin

---

Code 11 Validated Ledger $\mathcal{VL}$ (centralized)

```
1:  Init: S ← ∅
2:  receive (GET) from process p
3:          send response (GETRES, S) to p
4:  receive (APPEND, r) from process p
5:      if Valid(S‖r) then
6:          S ← S‖r
7:          send response (APPENDRES, ACK) to p
8:      else send response (APPENDRES, NACK) to p
```

---

protocol [21]. The ledger preserves the semantics with a validity check in the form of a Boolean function $Valid()$ that takes as an input a sequence of records S and returns *true* if and only if the semantics are preserved. In a validated ledger the result of an $append_p(r)$ operation may be NACK if the validity check fails. Code 11 presents a centralized implementation of a validated ledger $\mathcal{VL}$.

The sequential specification of a validated ledger must take into account the possibility that an append returns NACK. To this respect, property (2) of Definition 1 must be revised as follows:

Definition 5. *The sequential specification of a* **validated** *ledger* $\mathcal{VL}$ *over the sequential history* $H_{\mathcal{VL}}$ *is defined as follows. The value of the sequence* $\mathcal{VL}.S$ *is initially the empty sequence. If at the invocation event of an operation* π *in* $H_{\mathcal{VL}}$ *the value of the sequence in ledger* $\mathcal{VL}$ *is* $\mathcal{VL}.S = V$*, then:*

1. *if* π *is a* $\mathcal{VL}.get_p()$ *operation, then the response event of* π *returns* V*,*

2(a). if π is an $\mathcal{VL}$.append$_p(r)$ operation that returns ACK, *then* $Valid(V\|r) = true$ *and at the response event of π, the value of the sequence in ledger $\mathcal{VL}$ is $\mathcal{VL}.S = V\|r$, and*

2(b). if π is a $\mathcal{VL}$.append$_p(r)$ operation that returns NACK, *then* $Valid(V\|r) = false$ *and at the response event of π, the value of the sequence in ledger $\mathcal{VL}$ is $\mathcal{VL}.S = V$.*

Based on this revised notion of sequential specification, one can define the eventual, sequential and atomic consistent validated distributed ledger and design implementations in a similar manner as in Section 3.

It is interesting to observe that a validated ledger $\mathcal{VL}$ can be implemented with a regular ledger $\mathcal{L}$ if we are willing to waste some resources and accuracy (e.g., not rejecting invalid records). In particular, processes can use a ledger $\mathcal{L}$ to store *all* the records appended, even if they make the validity to be broken. Then, when the function get() is invoked, the records that make the validity to be violated are removed, and only the valid records are returned. The algorithm is presented in Code 12. This algorithm does not check validity in a $\pi = \text{append}(r)$ operation, which hence always returns ACK, because it is not possible to know when π is processed the final position r will take in the ledger (and hence to check its validity).

Code 12 Validated Ledger $\mathcal{VL}$ Implemented with a Ledger $\mathcal{L}$

```
 1:  Declare L: ledger
 2:  receive (GET) from process p
 3:      S ← ∅
 4:      V ← L.get()
 5:      foreach record r ∈ V do
 6:          If Valid(S‖r) then S ← S‖r
 7:      send response (GETRES, S) to p
 8:  receive (APPEND, r) from process p
 9:      res ← L.append(r)
10:      send response (APPENDRES, res) to p
```

---

5 Conclusions

In this paper we formally define the concept of a distributed ledger object with and without validation. We have focused on the definition of the basic operational properties that a distributed ledger must satisfy, and their consistency semantics. Finally, we have explored implementations of fault-tolerant distributed ledger objects with different types of consistency in crash-prone systems augmented with an atomic broadcast service.

As mentioned, this paper is only an attempt to formally address the many questions that were posed in the introduction. In that sense we have only scratched the surface. There is a large list of pending issues that can be explored. For instance, we believe that the implementations we have can be easily adapted to deal with Byzantine failures if the appropriate atomic broadcast service is used. Another extension worth exploring is how to deal with highly dynamic sets of possibly anonymous servers in order to implement distributed ledgers, to get closer to the Bitcoin-like ecosystem. In a more ambitious but possibly related tone, we would like to fully explore the properties of validated ledgers and their relation with cryptocurrencies.

References

1. Could blockchain provide the technical fix to solve sciences reproducability, crisis?

2. Namecoin. https://www.namecoin.org/.

3. Ittai Abraham, Dahlia Malkhi, et al. The blockchain consensus layer and bft. *Bulletin of EATCS*, 3(123), 2017.

4. Emmanuelle Anceaume, Romaric Ludinard, Maria Potop-Butucaru, and Frédéric Tronel. Bitcoin a distributed shared register. In *Stabilization, Safety, and Security of Distributed Systems - 19th International Symposium, SSS 2017, Boston, MA, USA, November 5-8, 2017, Proceedings*, pages 456–468, 2017.

5. Hagit Attiya and Jennifer L. Welch. Sequential consistency versus linearizability. *ACM Trans. Comput. Syst.*, 12(2):91–122, 1994.

6. Miguel Castro and Barbara Liskov. Proactive recovery in a byzantine-fault-tolerant system. In *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation-Volume 4*, page 19. USENIX Association, 2000.

7. Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, 20(4):398–461, 2002.

8. Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996.

9. Xavier Défago, André Schiper, and Péter Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, 36(4):372–421, 2004.

10. Juan A. Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol: Analysis and applications. In *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part II*, pages 281–310, 2015.

11. 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`, June 2017. Last accessed: 20 Oct. 2017.

12. Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 51–68. ACM, 2017.

13. Vincent Gramoli. From blockchain consensus back to byzantine consensus. *Future Generation Computer Systems*, 2017.

14. Vincent Gramoli. Securing sensitive data with the red belly blockchain. 2017.

15. Maurice Herlihy. Blockchains and the future of distributed computing. In Elad Michael Schiller and Alexander A. Schwarzmann, editors, *Proceedings of the ACM Symposium on Principles of Distributed Computing, PODC 2017, Washington, DC, USA, July 25-27, 2017*, page 155. ACM, 2017.

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

17. Tsung-Ting Kuo, Hyeon-Eui Kim, and Lucila Ohno-Machado. Blockchain distributed ledger technologies for biomedical and health care applications. *Journal of the American Medical Informatics Association*, 24(6):1211–1220, 2017.

18. Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

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

20. Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

21. Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008.

22. Nicolas Nicolaou, Antonio Fernández Anta, and Chryssis Georgiou. Cover-ability: Consistent versioning in asynchronous, fail-prone, message-passing environments. In *Network Computing and Applications (NCA), 2016 IEEE 15th International Symposium on*, pages 224–231. IEEE, 2016.

23. Nathaniel Popper and Steve Lohr. Blockchain: A better way to track pork chops, bonds, bad peanut butter? *New York Times*, March 2017.

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

25. Fred B Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319, 1990.

26. Avi Spielman. *Blockchain: Digitally Rebuilding the Real Estate Industry*. MS dissertation, Massachusetts Institute of Technology, 2016.

27. Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. `http://bitcoinaffiliatelist.com/wp-content/uploads/ethereum.pdf`, 2014. Accessed: 2016-08-22.