Received <day> <Month>, <year>; | Revised <day> <Month>, <year>; | Accepted <day> <Month>, <year>

DOI: xxx/xxxx

SPECIAL ISSUE PAPER

Eventually Consistent Distributed Ledger Despite Degraded Atomic Broadcast

Grégory Bénassy[1,2] | Fukuhito Ooshita[1] | Michiko Inoue*[1]

[1]Division of Information Science, Graduate School of Science and Technology, Nara Institute of Science and Technology, Nara, Japan
[2]Télécom SudParis, Evry, France

Correspondence
*Michiko Inoue, Takayama Ikoma, Nara, Japan. Email: kounoe@is.naist.jp

Abstract

The distributed ledger or blockchain technologies originated from the Bitcoin have been rapidly widespread in recent years. However, it also gives incentive to malicious users who would like to break the system or take advantage of it (steal money, hide some information stored in the ledger, isolate a particular node from the rest of the network, etc.). Thus, research focusing on overcoming potential attacks to distributed ledgers is required.
In this paper, we focus on attacks that damage underlying networks of distributed ledgers. Underlying networks offer useful communication primitives such as an atomic broadcast, however such attacks may degrade the property of the primitives and make distributed ledgers relying on the primitives no longer work. Hence we should design algorithms to make the distributed ledgers still work even when some attacks degrade the primitives.
As the first study for such situations, we consider a problem to implement distributed ledgers tolerating the degradation of an underlying atomic broadcast service that distributed ledgers are relying on. We consider the case where the uniform agreement property of the atomic broadcast is degraded, and propose new algorithms that could ensure to reach eventual consistency despite degraded atomic broadcast.

KEYWORDS:
distributed ledgers, atomic broadcast, eventual consistency.

1 | INTRODUCTION

Background. In 2009, Satoshi Nakamoto introduced Bitcoin[1], an open source peer-to-peer money. Since its creation and even today, Bitcoin is the main crypto-currency in the world. Indeed, blockchain technology[2] that makes Bitcoin system possible offers various advantages like openness, immutability, decentralization or the possibility of making transactions in a verifiable and permanent way without any central authority. Currently, the blockchain or distributed ledger technologies have been widespread beyond crypto-currencies or financial usage. The applications are extending to several domains including integrity verification, governance, IoT, healthcare management, privacy and security, supply chain management[3]. Thus, this huge hype around blockchain concept gives incentives to work on distributed computing[4] which is one of the building blocks of the technology with cryptography.

The wide usage of distributed ledger technologies also gives incentive to malicious users who would like to break the system or take advantage of it (steal money, hide some information stored in the ledger, isolate a particular node from the rest of the network, etc.). Even if blockchain systems are supposed to be unbreakable while trustful users control more than half of the computing power (51% attack), it is possible to hack such systems via, for example, Distributed Denial of Service (DDoS) attack, routing attack[5] and eclipse attack[6,7]. The eclipse attack consists in isolating a node from the rest of the network in order to perform a double-spend attack on this victim which does not know the real state of the blockchain and so is not aware that the amount of crypto-currency spent in the transaction has already been transferred before in another transaction. Heilman et al. carried out researches to show how to set up such a type of attack on the Bitcoin[6] and Ethereum[7] blockchains. Therefore, research focusing on overcoming potential attacks to distributed ledgers is required.

Related work. To exactly discuss the consistency property and fault-tolerance of distributed ledgers, formalization of distributed ledgers have been required. Many researches have been conducted to formalize distributed ledgers and their level of coherency[8,9,10]. A formalization of the behavior of distributed ledgers is given[8], where they introduced Distributed Ledger Register (DLR) as a distributed shared object that mimics Bitcoin. The level of consistency offered by Bitcoin, that cannot be explained by any classical shared objects, is captured in DLR. Distributed ledgers are formally specified with a combination of block tree and token oracle abstract data types, where the block tree formalizes the data structure and the token oracle captures the creation of blocks[9]. The formalization well captures the eventual convergence property that is specific to distributed ledgers. The mapping between the proposed abstraction and representative existing blockchains (including Bitcoin and Ethereum[11]) is shown.

Anta et al.[10] have formalized and implemented a way to reach eventual consistency on distributed ledgers using underlying networks which work with communication primitives such as atomic broadcast to maintain ledger coherency. They provided a formulation of a distributed ledger object that maintains a sequence of records through two types of operations Get and Append. Three consistency conditions, atomic consistency, sequential consistency and eventual consistency, are defined in terms of the operations supported by the distributed ledger object. Three implementations of distributed ledger objects satisfying these consistency conditions using atomic broadcast service are presented, where an underlying server network maintains a record sequence and clients access a distributed ledger object through communication with servers. As underlying systems, they consider asynchronous message passing systems with potential crash failures where atomic broadcast service is provided in the presence of crashed servers.

Our contribution. The economical aspects of distributed ledger technologies attract potential malicious users who would try to break the coherency. As it has been explained previously, nodes can be isolated from the network and fully controlled by adversary in the eclipse attack, but in this particular case, it is impossible to recover consistency because the node can communicate with only malicious users. For this reason, we consider a weaker attack, called partial eclipse attack, where the victim is surrounded by several attackers which try to hide some information (records, transactions, etc.) from the victim, yet, it is still able to communicate with correct nodes. We assume that these attacks damage underlying networks of distributed ledgers and so some properties of the atomic broadcast are degraded and do not stand anymore.

Hence, our work aims to find new algorithms based on the ones defined by Anta et al.[10] to overcome such a type of attacks that may degrade atomic broadcast properties. To do so, we first define the degradation of atomic broadcast service, where we consider the degradation of the uniform agreement property, one of four properties to define atomic broadcast service, in this paper. The algorithms for clients and servers that achieve eventual consistency under degraded atomic broadcast service are presented. This is an extended version of our previous work[12]. In this paper, we refine some algorithms and give formal proofs for all the proposed algorithms.

Paper Roadmap. The paper is organized as follows. In Section 2, some fundamental definitions are given about distributed ledgers, coherency and eventual consistency and atomic broadcast. Moreover, the model studied and used in the algorithms is described at the end of this part. Then, Section 3 presents the algorithms and the correctness of each one is proven. Section 4 discusses a relation between our approach and another approach that enhances atomic broadcast service so as to tolerate malicious attacks. Finally, Section 5 concludes the paper.

2 | PRELIMINARIES

In this subsection, we provide the fundamental definitions of distributed ledger, eventual consistency and atomic broadcast service, and introduce a degraded atomic broadcast service that is a target to overcome in this paper.

2.1 | Shared Object and Distributed Ledger

We consider a shared object accessed by multiple clients. A shared object is specified by the set of values that the object can take and the set of operations to access the values. A ledger object (or simply ledger) L is a shared object that stores a sequence of records which corresponds to the state of the ledger and they support two particular operations: the Get operation to get the sequence stored in the ledger and the Append operation to add a record to the sequence as it is defined in[10] (it is equivalent to the read and write operations from[8]). Multiple clients can concurrently access to the ledger using these operations. Each operation consists of two events: *invocation* and *response*, and they occur in this order. When a client accesses to the ledger with some operation, the client *invokes* the operation (invoke event) and then the ledger *responses* to the client (response event). A *history* H_L of a ledger L is a sequence of operation events and the order in such history corresponds to the real-time order of the events. An operation is *complete* when its invocation and response events have occurred. A history H_L is *complete* if it is composed of only complete operations. Two operations are *concurrent* if the second one is invoked before the first one reaches its response event. A complete history

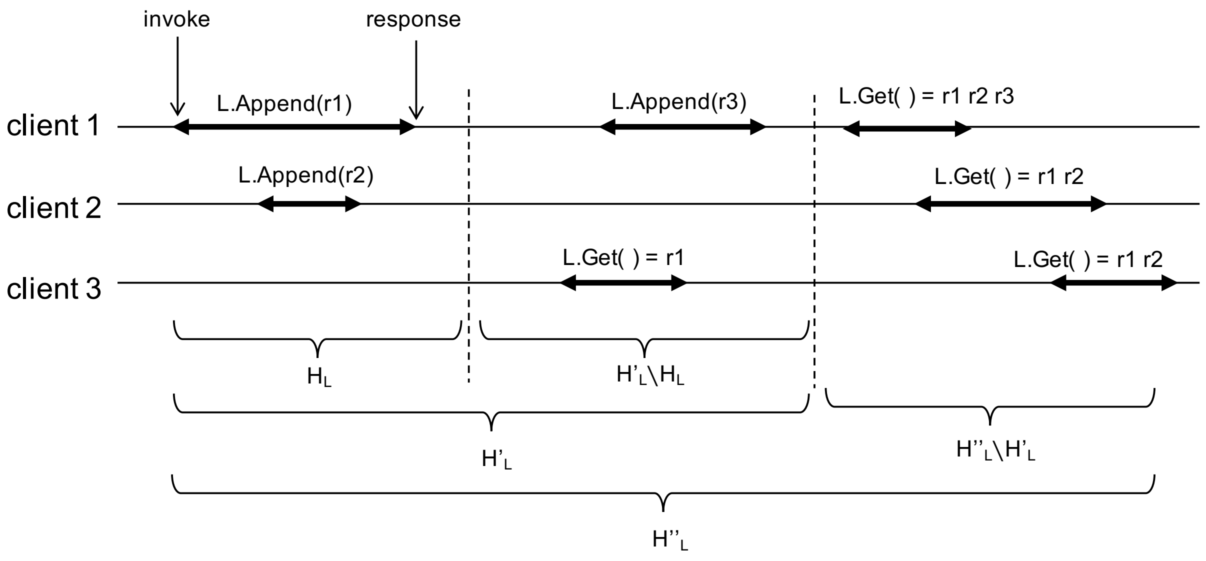

FIGURE 1 Illustration of definition 2 : operations invoked by three clients are illustrated, where the left and right edges of the arrow representing each operation show the timings for its invoke and response, respectively. r_i are records appended by the clients and H_L, H'_L and H''_L are histories.

is *sequential* if it is composed of no concurrent operations. Then, we can define the *sequential specification* of a ledger L (from[10]), which shows the behavior of L when it is accessed sequentially.

Definition 1. [10] The **sequential specification** of a ledger L over the sequential history H_L is defined as follows. The value of the sequence L.S of the ledger is initially an empty sequence. If at the invocation event of an operation π in H_L the value of the sequence in ledger L is $L.S = V$, then:

1. if π is an $L.\mathsf{Get}()$ operation, then the response event of π returns V, and
2. if π is an $L.\mathsf{Append}(r)$ operation, at the response event of π, the value of the sequence in ledger L is $L.S = V||r$ (where $||$ is the concatenation operator).

A distributed ledger is a ledger object implemented in a distributed system with multiple servers. A distributed ledger is implemented with two type algorithms for clients and servers respectively where operations are implemented with interactions between a client and servers. The implementation guarantees some consistency. As a consistency condition for a distributed ledger, atomic consistency, sequential consistency, and eventual consistency are considered.

2.2 | Eventual Consistency

Eventual consistency is one of the weakest consistency condition for distributed ledgers. Indeed, it only has to eventually converge which means that after a certain amount of time all clients get the same sequence from the network. With eventual consistency, servers may disagree. In this paper, we focus on the eventual consistency and follow the definition given in the paper[10]:

Definition 2. [10] A distributed ledger L is eventually consistent if, given any complete history H_L, there exists a permutation σ of the operations in H_L such that:

1. σ follows the sequential specification of L, and
2. there exists a complete history H'_L that extends H_L such that, for every complete history H''_L that extends H'_L, every complete operation $L.\mathsf{Get}()$ in $H''_L \backslash H'_L$ returns a sequence that contains r, for every $L.\mathsf{Append}(r) \in H_L$.

Figure 1 illustrates this definition. A complete history H_L is composed of operations $L.\mathsf{Appended}(r1)$ and $L.\mathsf{Appended}(r2)$, and a complete history H'_L is composed of $L.\mathsf{Appended}(r1)$, $L.\mathsf{Appended}(r2)$, $L.\mathsf{Appended}(r3)$, and $L.\mathsf{Get}()$. Focusing on H_L in this figure, there is a permutation of two operations in H_L that follows the sequential specification of a ledger (two Append can be ordered arbitrarily), and after completing H'_L (that is, in $H''_L \backslash H'_L$), every $\mathsf{Get}()$ operation returns a sequence including both r1 and r2.

Though the definition of the eventual consistency is discussed for only complete history in this paper, it can be extended to include non-complete (or partial) history $H_L^{partial}$ where some invocations do not have their responses as discussed in[13,14]. A partial history $H_L^{partial}$ is eventually consistent if it can be modified to some complete history H_L by deleting some invocations and appending some responses, and H_L satisfies the condition of the eventual consistency.

2.3 | Atomic Broadcast

The atomic broadcast service is a fundamental primitive to deliver messages to all the nodes in a distributed system in the same order. Implementations of distributed ledgers relying on the atomic broadcast service are proposed[10], where the following definition[15] is adopted.

Definition 3. In fault-tolerant distributed computing, atomic broadcast is a broadcast where all the nodes of the network deliver the same messages in the same order. To do so, four properties are required:

- Validity: if a correct server broadcasts a message, then the server will eventually deliver the message.
- Uniform Agreement: if a correct server delivers a message, then all the correct servers will eventually deliver it.
- Uniform Integrity: a message is delivered by each server at most once, and only if it has been previously broadcasted.
- Uniform Total Order: if a server delivers message m before message m', then, all the servers that deliver those messages must do it in the same order.

In this paper, the goal is to reach eventual consistency even in the case of victim servers that do not manage to deliver some messages.

2.4 | Model

Our model of distributed systems is based on the model defined in[10]. We consider an asynchronous distributed system with a set S of n multiple servers, where an atomic broadcast service is provided as a primitive. In the model, some servers can simply crash and get *faulty* but no more than $f < n/2$, where f is an upper bound of faulty servers. It is known that atomic broadcast could not be deterministically solved in asynchronous systems if there is at least one crashed server[16,17]. The condition $f < n/2$ is required for the atomic broadcast service to work properly when failure detectors with eventual accuracy[16] is available. This implies that we implicitly assume failure detectors or similar enhancement. A server can get faulty during an execution, and the faulty server does not do anything once it becomes faulty. We assume that the clients know the upper bound f of faulty servers. In the definition of atomic broadcast (Definition 3), a correct server means a server that is not faulty. That is, servers are classified into correct and faulty.

In this paper, we consider a scenario where an atomic broadcast service that works properly in presence of faulty servers is degraded due to some malicious attack to underlying networks. To represent such a degradation, in our model, we consider some potential *victim servers* that do not deliver some messages, which is a violation of the *uniform agreement* property of the atomic broadcast. This is what we call a degraded atomic broadcast service. In our model, the *uniform agreement* property of the atomic broadcast is replaced with the following *degraded uniform agreement* property while other three properties still hold.

- Degraded Uniform Agreement: if a correct server delivers a message, then all the correct and non-victim servers will eventually deliver it.

When a victim server fails to deliver a record, its maintained sequence of records becomes wrong. We assume at most m *victim servers* for each broadcast. A victim server may change over time and hence every server can become a victim server, and may keep wrong sequences of records.

Clients and servers communicate through asynchronous reliable channels. There is no limit of clients that can access the distributed ledger.

3 | PROPOSED ALGORITHMS

3.1 | Algorithm based on majority voting

In this subsection, we propose algorithms to implement a distributed ledger. The algorithms decide a return value of Get operation based on majority voting. We assume that $n \geq f + 2m + 1$ holds, where n, f, and m are the number of servers, the upper bounds of faulty servers and victim servers, respectively. Algorithm 1 is for clients while Algorithm 2 is for servers. Algorithm 2 is the algorithm proposed to implement eventual consistent distributed ledger in case of no victim server[10].

Algorithm 1 Majority voting (Code for client p)

```
 1: Init
 2:     c ← 0
 3: function L.Get( )
 4:     Let $K \subseteq S$: $|K| \geq f + 2m + 1$
 5:     $c \leftarrow c + 1$
 6:     send request (c, Get) to the servers in K
 7:     wait response (c, GetRes, $V_i$) from $2m + 1$ servers $i \in K$
 8:     $V \leftarrow [V_1, ..., V_{2m+1}]$
 9:     return combine (V)
10: function L.Append(r)
11:     Let $K \subseteq S$: $|K| \geq f + 1$
12:     $c \leftarrow c + 1$
13:     send request (c, Append, r) to the servers in K
14:     wait response (c, AppendRes, res) from some $i \in K$
15:     return res
16: function combine(V)
17:     Seq ← $\emptyset$
18:     if $|V_i| = 0$ then remove $V_i$ from V
19:     $\forall i \in [1, 2m + 1]$ $ind_i \leftarrow 0$
20:     while $V \neq \emptyset$ do
21:         RecordList = a list of $V_i(ind_i) \in V$
22:         if $\exists$ R appears at least $m + 1$ times in RecordList then
23:             Seq ← Seq||R
24:         else
25:             break
26:         for $i \leftarrow 1$ to $2m + 1$ do
27:             while $V_i \in V$ and $V_i(ind_i) \in Seq$ do
28:                 $ind_i = ind_i + 1$
29:             if $ind_i = |V_i|$ then
30:                 remove $V_i$ from V
31:     return Seq
```

Algorithm 2 (from [10]) Majority voting (Code for server i)

```
 1: Init
 2:     $Seq_i \leftarrow \emptyset$

 3: if receive (c, Get) from client p then
 4:     send response (c, GetRes, $Seq_i$) to p
 5: if receive (c, Append, r) from client p then
 6:     ABroadcast(r)
 7:     send response (c, AppendRes, ACK) to p
 8: if ADeliver(r) then
 9:     if $r \notin Seq_i$ then
10:         $Seq_i \leftarrow Seq_i||r$
```

For Get operation, a client sends requests to a set of $|K|$ servers when invoking an operation. For Get operation, $|K|$ must be $f + 2m + 1$ or more because at least $2m + 1$ server responses are required even in the worst case where the number of faulty servers reaches the upper bound f. When receiving the request for Get operation, each server sends a sequence of records that the server locally maintains as a response. The client waits for $2m + 1$ responses for the invoked Get operation. Finally, the combine function is used to rebuild a sequence of records that every server can have if there is no victim server. $V_i(ind_i)$ used in the function combine means the ind_i-th record in a sequence V_i. Each operation is distinguished by a counter value c which is incremented each time a client executes an operation. Therefore, clients can wait for the right amount of messages and ignore the additional ones.

For Append(r) operation, a client sends requests to append a record r to a set of $|K|$ servers. In this case, $|K|$ must be $f + 1$ or more since only one response is required to guarantee the termination of the operation. When receiving the request for Append(r) operation, each server broadcasts a record r using the atomic broadcast service and then sends the response to the client. Once the client receives one response from a server, it completes the operation.

Append(r) operation terminates after confirming some server broadcasts a message containing a record r to servers, and the message will be delivered to all correct and non-victim servers. Finally, when a server delivers a broadcast message, it first checks if the record already belongs to its local sequence and if it is not, the record is appended to its own local sequence. ABroadcast(r) and ADeliver(r) are underlying functions from the atomic broadcast service to add the records complying with all the properties defined previously in section 2.3.

To prove the correctness, we first define some terminology. In an execution of a combination of Algorithm 1 (client) and Algorithm 2 (server), all the servers deliver messages in the same order if there is no victim server, that is, they receive records in the same order. We call the order of records in case of no victim server *a potential delivered record sequence*. In Algorithm 1, a client sends requests to multiple servers considering faulty servers. So, it is possible that multiple correct servers receive and broadcast the same records and then they are delivered. When multiply delivering the same record, the server appends it to its Seq only when it is delivered first time. The sequence of records obtained from the potential delivered record sequence by extracting the first entry for each record is called *a potential appended record sequence*. Note that any prefixes of a potential delivered record sequence and a potential appended record sequence are also a potential delivered record sequence and a potential appended record sequence, respectively. If a record appears in a potential appended record sequence, it is called *true* record, otherwise the record

potential delivered record sequence	$\mathbf{r_1}$, $\mathbf{r_2}$, r_1, $\mathbf{r_3}$, r_2, r_2, r_1, $\mathbf{r_4}$, $\mathbf{r_5}$, r_3, r_2, r_4, $\dots$
potential appended record sequence	$\mathbf{r_1}$, $\mathbf{r_2}$, $\mathbf{r_3}$, $\mathbf{r_4}$, $\mathbf{r_5}$, $\dots$
delivered record sequence for a victim server	$\mathbf{r_1}$, $\mathbf{r_2}$, r_1, r_2, r_2, r_1, $\mathbf{r_4}$, $\mathbf{r_5}$, r_3, r_2, r_4, $\dots$
local record sequence for a victim server	$\mathbf{r_1}$, $\mathbf{r_2}$, $\mathbf{r_4}$, $\mathbf{r_5}$, r_3, $\dots$

FIGURE 2 Delivered record sequence and appended record sequence. Bold font means true records, while italic font means false records.

is called *false* record. Note that true record and its false record are the same record, but we distinguish these records in the discussion in our proofs since they are broadcasted by different servers. Each server maintains a sequence of records, we call it *local record sequence*. If there is a victim server, it may fail to deliver some true records. In that case, some true record is missing from a delivered record sequence for the victim server, and some false record may appear in its local record sequence. Figure 2 shows an example for these sequences.

Let V_{pd} denote a potential delivered record sequence, and Let $V(ind)$ denote the ind-th record in a sequence V. For example, letting V_i be a local record sequence for a victim server in Figure 2 , $V_i(2) = V_{pd}(7) = r_4$ (indexes start with 0). Let $Index_i(r)$ and $Index_{pd}(r)$ denote indexes of r in V_i and V_{pd}, respectively.

Lemma 1. A local record sequence for each server is a subsequence of a potential delivered record sequence.

Proof. Since there is at most m victim servers for each broadcast and there are at least $2m+1$ correct servers, for any pair of two messages, at least one server can deliver the both messages. Therefore, any pair of two messages has their order and it is common to any server from the uniform total order property. That implies that there is an total order among messages, and every delivered record sequence follows the total order. Since a local record sequence for each server is a subsequence of its delivered record sequence, and it is also a subsequence of a potential delivered record sequence. □

Lemma 2. Function $combine(V)$ at line 9 in Algorithm 1 returns a potential appended record sequence.

Proof. In Algorithm 1, when invoking $L.Get()$, a client waits for $2m+1$ responses from servers, and then gets $2m+1$ local sequences of records $V_1, V_2, \dots, V_{2m+1}$. In these $2m+1$ sequences, false records appear in at most m sequences for each corresponding true record since there is at most m victim server for a broadcast that fails to deliver the true record. In addition, for a true record r^{true} and its corresponding false record r^{false}, $index_{pd}(r^{true}) < index_{pd}(r^{false})$ holds.

We will show by induction that, in each iteration of the outer while loop in $combine()$, one true record is appended to the final result Seq in the order of the potential appended sequence or the iteration is terminated by a *break* statement. Note that, if ind_i exceeds the length of V_i, $RecordList$ does not include a record from V_i.

(base step) Each $V_i(0)$ is a true record $V_{pd}(0)$ if the corresponding server successfully delivered $V_{pd}(0)$. That means, in the first iteration, there are at most m records other than $V_{pd}(0)$ in $RecordList$, and only $V_{pd}(0)$ has a chance to appear at least $m+1$ times in $RecordList$. Therefore, $V_{pd}(0)$ is successfully added to Seq or the iteration is terminated by a *break* statement. At the end of this iteration, each ind_i is increased until it indicates a record that has not been appended to Seq or reaches the end of the sequence.

(induction step) Assume that some true records are appended to Seq in the order of the potential appended sequence, any true record is not skipped to be appended, and each $V_i(ind_i)$ in $RecordList$ is a record that has not been appended to Seq. If any record appears at most m times in $RecordList$, the iteration is terminated. Consider the case where some record appears at least $m+1$ times in $RecordList$. If there is a false record r^{false} in $RecordList$, that implies the corresponding true record r^{true} has not been appended to Seq, and therefore, there is a record r in $RecordList$ that precedes (or is equal to) r^{true} in some local record sequence. From Lemma 1, $Index_{pd}(r) \leq Index_{pd}(r^{true}) < Index_{pd}(r^{false})$ holds. Let r_{min} be a record with the minimum index in V_{pd} among records in $RecordList$. From the above discussion, r_{min} is a true record. Since r_{min} has the minimum index in V_{pd} among records in $RecordList$, true records that have less index than r_{min} have been appended in Seq. That is r_{min} is the first true record that has not been appended in Seq in the potential appended record list. Since a true record is failed to be delivered at most m servers, there are at most m records other than r_{min} in $RecordList$. Therefore, a record that appears at least $m+1$ in $RecordList$ is r_{min}, and r_{min} is appended to Seq. At the end of this iteration, each ind_i is increased until it indicates a record that has not been appended to Seq or reaches the end of the sequence. In this process, false records can be skipped if the corresponding true record has been already appended. □

Theorem 1. Combination of Algorithm 1 (client) and Algorithm 2 (server) implements an eventually consistent distributed ledger in the case where a record can be failed to be delivered by at most m victim servers for each broadcast.

Algorithm 3 Shortest common supersequence (Code for client p)

```
1: Init
2:     $c \leftarrow 0$
3: function L.Get( )
4:     Let $K \subseteq S$: $|K| \geq f + m + 1$
5:     $c \leftarrow c + 1$
6:     send request (c, Get) to the servers in K
7:     wait response (c, GetRes, $V_i$) from $m + 1$ servers ($1 \leq i \leq m + 1$)
8:     Seq $\leftarrow$ remove-false(scs(remove-pending($V_1, \ldots, V_{m+1}$)))
9:     return Seq
```

```
1: function L.Append(r)
2:     Let $K \subseteq S$: $|K| \geq f + 1$
3:     $c \leftarrow c + 1$
4:     send request (c, Append, r) to the servers in K
5:     wait response (c, AppendRes, res) from some $i \in K$
6:     return res
```

Algorithm 4 Shortest common supersequence (Code for server i)

```
1: Init
2:     $Seq_i[j] \leftarrow \emptyset$ for $\forall j \in [1, n]$
3: if receive (c, Get) from client p then
4:     send response (c, GetRes, $Seq_i[i]$) to p
5: if receive (c, Append, r) from client p then
6:     ABroadcast(r, $Seq_i[i]$)
7:     send response (c, AppendRes, ACK) to p
```

```
8: if ADeliver(r, Seq) from server j then
9:     if $r \notin Seq_i[i]$ then
10:        $Seq_i[i] \leftarrow Seq_i[i] || r$
11:    if $|Seq| > |Seq_i[j]|$ then $Seq_i[j] \leftarrow Seq$
12:    $Seq_i[i] \leftarrow$ merge($Seq_i[i], Seq_i[1], \ldots, Seq_i[n]$)
```

Proof. Let us consider any complete history H_L and some victim servers that have missed some records. First, we show that operations in H_L can be permutated so as to satisfy the sequential specification. In L.Append(r) operation, a client sends request to append r to $f + 1$ or more servers, and then it will be received by one or more correct servers and r is broadcasted by these servers. That is, all the records in L.Append() operations in H_L is broadcasted by some correct server. On the other hand, in L.Get() operation, a client sends request to get a sequence of records to $f + 2m + 1$ or more servers, and waits for $2m + 1$ responses. Then, the client combines the $2m + 1$ sequences by function combine() and returns the obtained sequence. Now we permutate all the L.Append() operations in the order of the potential appended record sequence, and then insert each L.Get() in an appropriate position as follows. Let L.Get() return Seq $= r_1, r_2, \ldots, r_i$. L.Get() is inserted after L.Append() that appends r_i and before the next L.Append() if exist. From Lemma 2, L.Get() returns a potential appended record sequence, and hence, the permutation follows the sequential specification.

Next, we show that every appended record in H_L eventually appears in sequences returned by L.Get() operations. In L.Append(r) operation, r is broadcasted by some correct server and it is eventually delivered by all the correct and non-victim servers from the validity and uniform agreement properties. Therefore, r is eventually appended in a local sequence of correct and non-victim servers. If clients repeatedly invoke L.Get(), eventually at least $m + 1$ sequences from the $2m + 1$ sequences that the client receives from servers contain r as a correct record and hence r appears in a returned sequence. That is, there exists a complete history H'_L that extends H_L such that, for every complete history H''_L that extends H'_L, every complete operation L.Get() in $H''_L \backslash H'_L$ returns a sequence that contains r, for every L.Append(r) $\in H_L$. □

3.2 | Algorithm based on shortest common supersequence

In Algorithm 1, clients send requests to at least $f + 2m + 1$ servers and then waits for $2m + 1$ responses to rebuild the right sequence from possibly wrong sequences. However, it seems costly in a practical view point. If the number of servers that have wrong sequences is bounded, the right sequence can be rebuilt without majority voting, and clients may reduce the number of requests.

For this motivation, we consider Algorithm 3 for clients and Algorithm 4 for servers with additional assumption on liveness. We first assume $n \geq f + m + 1$, and later we discuss the required number of servers for additional liveness property. Consider an execution where Append() operations are invoked infinitely often, and every correct server periodically has chances to get Append requests and broadcast the requests to other servers. In this situation, if servers also broadcast their own local record sequences with records to be appended, servers can correct their local record sequences upon delivering the broadcast messages. In this subsection, we propose another combination of algorithms to implement a distributed ledger with some assumption on liveness. Since the assumption is on an execution of the proposed algorithm, we first describe the algorithms.

The algorithms have two strategies:

1. When a server broadcasts a new record, the server attaches its local record sequence.
2. A client rebuilds a record sequence from multiple sequences based on a shortest common supersequence.

In Algorithm 4, each server i stores record sequences $\mathsf{Seq}_i[j]$ broadcasted from other servers j as well as its local record sequence $\mathsf{Seq}_i[i]$. To exactly distinguish true and false records, a record is identified with its content r and a server i that broadcasted the record as (r, i). When a server i broadcasts a new record (r, i), it also broadcasts its local sequence. When server i delivers a record sequence Seq broadcasted from server j, server i corrects its local sequence according to Seq.

The local sequence is corrected by a function $\mathsf{merge}()$. Function $\mathsf{merge}(\mathsf{Seq}, \mathsf{Seq}_1, \mathsf{Seq}_2, \ldots)$ returns a corrected sequence of the first argument Seq from the following sequences $\mathsf{Seq}_1, \mathsf{Seq}_2, \ldots$ by inserting missing records and then removing false records. For example, if $\mathsf{Seq} = (r_1, i)(r_3, j)(r_2, k)$ and $\mathsf{Seq}_1 = (r_1, i)(r_2, l)(r_3, j)$ are given, (r_2, l) is inserted between (r_1, i) and (r_3, j) and then (r_2, k) is identified to be false and removed. Consequently, $\mathsf{Seq} = (r_1, i)(r_2, l)(r_3, j)$ is obtained. On the other hand, if $\mathsf{Seq} = (r_1, i)(r_2, j)(r_4, k)$ and $\mathsf{Seq}_1 = (r_1, i)(r_3, l)(r_4, k)$ are given, we cannot determine the position of (r_3, l) and it is not inserted. However, in this case, if there is another sequence $\mathsf{Seq}_2 = (r_2, j)(r_3, l)(r_4, k)$, we can resolve the order of (r_2, j) and (r_3, l), (r_3, l) is inserted to Seq, and $\mathsf{Seq} = (r_1, i)(r_2, j)(r_3, l)(r_4, k)$ is obtained. In Algorithm 4, servers can exchange their local sequences so that they are merged into a correct sequence. As we showed in Lemma 1, each local record sequence is a subsequence of a potential delivered record sequence. Therefore, local sequences have consistent orders and no conflict occurs when merged.

In Algorithm 3 for clients, a client sends requests to at least $f + m + 1$ servers and waits for $m + 1$ responses. The right sequence can be rebuilt based on a shortest common supersequence from multiple local sequences. A shortest common supersequence SCS for multiple sequences $\mathsf{LS}_1, \mathsf{LS}_2 \ldots$ is the shortest sequence among sequences to have each LS_i as a subsequence. For example, a shortest common supersequence of "a b c e" and "a c d e" is "a b c d e". In general, a shortest common supersequence is not unique. For example, for two sequences "a b c e" and "a b d e", both "a b c d e" and "a b d c e" are shortest common supersequences since we could resolve an order of c and d. In Algorithm 3, a function scs returns a maximal prefix of a shortest common supersequence in which a total order among records is resolved. For example, scs("a b c e", "a b d e") returns "a b".

The final sequence is obtained from the shortest common supersequence with removing two types of records: false and pending. False records can be included in the shortest common supersequence. Since every false record always appears after its true record, it is easily removed. For a set of local record sequences, there may be records that do not appear in all the sequences. Such records are failed to delivered, or will be eventually delivered (i.e., have not been delivered). A record that might be delivered is called *pending* record, and defined as follows. For a set of local record sequences S, we say a record (r, i) precedes another record (r', i'), if (r, i) precedes (r', i') in some local record sequence or there is a record (r'', i'') such that (r, i) precedes (r'', i'') and (r'', i'') precedes (r', i'). A record (r, i) is pending in a set of local record sequences V if there is a local record sequence $\mathsf{LS} \in V$ such that (1) (r, i) does not appear in LS and (2) there is no record (r', i') in LS such that (r, i) precedes (r', i') in V. For example, consider a set of three local record sequences $\mathsf{Seq}_1 = (r_1, i)\ (r_2, j)$, $\mathsf{Seq}_2 = (r_2, j)\ (r_3, k)$, $\mathsf{Seq}_3 = (r_3, k)$. In this case, (r_1, i) precedes (r_2, j), (r_2, j) precedes (r_3, k), and then (r_1, i) precedes (r_3, k). Therefore, (r_1, i) or (r_2, j) are not pending, while (r_3, k) is pending. We cannot include a record (r_3, k) in a response of $\mathsf{Get}()$ operation since some record may precedes (r_3, k) in a potential appended record sequence but it does not appear in a set of local record sequences.

Next lemma shows a combination of the algorithms return correct responses. Let us consider any complete history H_L for a distributed ledger L implemented by Algorithms 3 and 4.

Lemma 3. There exists a permutation σ of the operations in H_L such that σ follows the sequential specification of L.

Proof. We permutate all the $L.\mathsf{Append}()$ operations in the order of the potential appended record sequence, and then insert each $L.\mathsf{Get}()$ in an appropriate position.

Let $L.\mathsf{Get}()$ return $\mathsf{Seq} = r_1, r_2, \ldots, r_i$. $L.\mathsf{Get}()$ is inserted after $L.\mathsf{Append}()$ that appends r_i and before the next $L.\mathsf{Append}()$ if exist. In Algorithm 4, a local record sequence of some server is repeatedly merged with local record sequences of other servers. However, Lemma 1 still holds since each merge combines subsequences of a potential delivered record sequence without changing the order of records and get a subsequence of the potential delivered record sequence. In $L.\mathsf{Get}()$ operation, a set of $m + 1$ local record sequences V is first obtained, and the final sequence Seq is obtained from V by removing pending records, getting a (prefix of) shortest common supersequence, and removing false records. Let SCS be a sequence returned from function scs. Since local record sequences are subsequences of a potential delivered record sequence, SCS is also a subsequence of the potential delivered record sequence.

We will show SCS includes r_i and all its preceding true records in the potential delivered record sequence. If some such a true record r' is missing in a local record sequence of some server, r' is never delivered by that server from uniform total order property since its succeeding record is already delivered (from the definition of pending record). Since there are at most m victim servers for each record and SCS is created from $m + 1$ local record sequences, at least one server delivers r' and it is included in SCS. After removing false records from SCS, we can obtains a sequence of true records up to r_i in the order of the potential appended record sequence without skipping any records. That is, the permutation follows the sequential specification. $\square$

To ensure the progress, now we consider an additional assumption on liveness. We consider only infinite execution and assume that every correct server executes $\mathsf{ABroadcast}()$ infinitely often and $n \geq f + 2m + 1$ holds. Under this additional assumption, a combination of Algorithms 3 and 4 can work correctly.

Theorem 2. A combination of Algorithms 3 and 4 implements an eventually consistent distributed ledger L if every correct server executes $\mathsf{ABroadcast}()$ infinitely often and $n \geq f + 2m + 1$ holds.

Proof. We show that every appended record in H_L eventually appears in sequences returned by $L.\mathsf{Get}()$ operations. In $L.\mathsf{Append}(r)$ operation, r is broadcasted by some correct server and it is eventually delivered by all the correct and non-victim servers from the validity and uniform agreement properties. Therefore, r is eventually appended in a local record sequence of correct and non-victim servers.

In addition, there are at least correct $2m + 1$ servers and therefore any pair of two true records r_1 and r_2 are delivered by some correct server. From the assumption, the server eventually broadcasts an order of r_1 and r_2, and at most m servers fail to deliver this information. This implies that any set of $m + 1$ servers eventually get enough information to resolve the order of any pair of r_1 and r_2, and hence get enough information to resolve total order up to r, and r appears in a returned sequence.

That is, there exists a complete history H'_L that extends H_L such that, for every complete history H''_L that extends H'_L, every complete operation $L.\mathsf{Get}()$ in $H''_L \backslash H'_L$ returns a sequence that contains r, for every $L.\mathsf{Append}(r) \in H_L$. Combining with Lemma 3, the theorem holds. $\square$

4 | DISCUSSION

In this paper, we are considering how to implement a distributed ledger when underlying atomic broadcast service is degraded due to malicious attacks. To compensate the degradation of atomic broadcast, two algorithms have been proposed. As a countermeasure for malicious attacks, we can also consider to enhance atomic broadcast service itself. If atomic broadcast is available despite malicious attacks, a distributed ledger using atomic broadcast as underlying communication can also work without modification. In this section, we will discuss a relation between these two approaches.

Atomic broadcast in asynchronous systems is well studied, and it is known that consensus and atomic broadcast are equivalent in terms of solvability in asynchronous systems with crash failures[16]. It is also well known that consensus and hence atomic broadcast could not be deterministically solved in asynchronous systems if there is at least one crashed process[17]. So, atomic broadcast in presence of failures has been studied to implement it using failure detectors or probabilistically[16,18,19]. In our model, we assume an atomic broadcast service that can work properly in presence of crash processes if there is no malicious attack. This implies that we implicitly assume failure detectors or similar enhancement.

Unreliable failure detectors are introduced, and algorithms for consensus and atomic broadcast using the failure detectors are demonstrated[16]. A failure detector is a special module that maintains a set or processes suspected to be crashed with possible mistakes. Rodrigues and Raynal enhance this work to consider crash-recovery failure model[18]. Crash-recovery failure model allows processes to crash and then recover. This model seems to be similar to our model where victim servers fail to deliver messages but they can rejoin the atomic broadcast service. This suggests to us that if we can model the behavior of attacks at a layer where an atomic broadcast service is implemented, we may contain the behavior of malicious attacks in an atomic broadcast service and we do not need to consider victim servers to implement a distributed ledger. However, their algorithm[18] is based on gossiping where processes repeatedly exchange information to guarantee some liveness property. That is, the consideration into malicious attacks in atomic broadcast services may need high communication cost.

Another approach to implement an atomic broadcast service in presence of failures considers probabilistic algorithms. Felber and Pedone propose a probabilistic atomic broadcast algorithm tolerating message loses and crashed processes[19]. The proposed algorithm includes some randomized operations, and due to the probabilistic nature of the algorithm, the requirements for atomic broadcast are modified. They consider *Probabilistic Agreement*, *Probabilistic Order*, and *Probabilistic Validity*. Probabilistic Agreement requires that if a correct server delivers a message, then any other correct server delivers it with some probability. This can be seen as a kind of degradation, and Probabilistic Agreement has a similar direction of our Degraded Uniform Agreement. It is also interesting to implement a distributed ledger using probabilistic atomic broadcast.

5 | CONCLUSION

In this paper, we have designed new two algorithms to implement eventually consistent distributed ledgers despite a degraded atomic broadcast service. In the proposed algorithms, a client rebuilds a correct record sequence from multiple local record sequences collected from multiple servers. The first one uses a majority voting that requires $2m + 1$ local record sequences. The second one uses a shortest common supersequence with an additional liveness assumption and reduces communication between clients and servers where a correct record sequence can be rebuilt from only $m + 1$ local record sequences.

In this paper, execution time has not been taken into account. It would be interesting as a future work to see the performance of algorithms to achieve the eventual consistency. Moreover, this paper only described some particular situations and many other issues can be explored. For instance, atomic broadcast can be weakened even more by violating some other properties like validity or uniform integrity. Also, reaching eventual consistency with a basically weaker broadcast like reliable broadcast that does not guarantee total order can be studied.

References

1. Nakamoto S. Bitcoin: A peer-to-peer electronic cash system. https://bitcoin.org/bitcoin.pdf; 2008.
2. Dumas JG, Lafourcade P, Tichit A, Varette S. *Les blockchains en 50 questions*. Dunod . 2018.
3. Casino F, Dasaklis TK, Patsakis C. A systematic literature review of blockchain-based applications: current status, classification and open issues. *Telematics and Informatics* 2019; 36: 55–81.
4. Attiya H, Welch J. *Distributed computing: fundamentals, simulations, and advanced topics*. 19. John Wiley & Sons . 2004.
5. Apostolaki M, Zohar A, Vanbever L. Hijacking bitcoin: Routing attacks on cryptocurrencies. In: 2017 IEEE Symposium on Security and Privacy (SP). IEEE. ; 2017: 375–392.
6. Heilman E, Kendler A, Zohar A, Goldberg S. Eclipse attacks on bitcoin's peer-to-peer network. In: 24th USENIX Security Symposium (USENIX Security 15). ; 2015: 129–144.
7. Marcus Y, Heilman E, Goldberg S. Low-Resource Eclipse Attacks on Ethereum's Peer-to-Peer Network.. *IACR Cryptology ePrint Archive* 2018; 2018: 236.
8. Anceaume E, Ludinard R, Potop-Butucaru M, Tronel F. Bitcoin a distributed shared register. In: International Symposium on Stabilization, Safety, and Security of Distributed Systems. Springer. ; 2017: 456–468.
9. Anceaume E, Del Pozzo A, Ludinard R, Potop-Butucaru M, Tucci-Piergiovanni S. Blockchain abstract data type. In: The 31st ACM Symposium on Parallelism in Algorithms and Architectures. ACM. ; 2019: 349–358.
10. Anta AF, Konwar K, Georgiou C, Nicolaou N. Formalizing and implementing distributed ledger objects. *ACM SIGACT News* 2018; 49(2): 58–76.
11. Wood G. Ethereum: A secure decentralised generalised transaction ledger. Ethereum project yellow paper; 2014.
12. Bénassy G, Ooshita F, Inoue M. Eventually Consistent Distributed Ledger Relying on Degraded Atomic Broadcast. In: 2019 Seventh International Symposium on Computing and Networking Workshops (CANDARW). ; 2019: 195-200.
13. Herlihy MP, Wing JM. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 1990; 12(3): 463–492.
14. Raynal M. *Concurrent programming: algorithms, principles, and foundations*. Springer Science & Business Media . 2012.
15. Défago X, Schiper A, Urbán P. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Surveys (CSUR)* 2004; 36(4): 372–421.
16. Chandra TD, Toueg S. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM (JACM)* 1996; 43(2): 225–267.
17. Fischer MJ, Lynch NA, Paterson MS. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)* 1985; 32(2): 374–382.
18. Rodrigues L, Raynal M. Atomic broadcast in asynchronous crash-recovery distributed systems and its use in quorum-based replication. *IEEE Transactions on Knowledge and Data Engineering* 2003; 15(5): 1206–1217.
19. Felber P, Pedone F. Probabilistic atomic broadcast. In: 21st IEEE Symposium on Reliable Distributed Systems, 2002. IEEE. ; 2002: 170–179.