Consensus-Free Ledgers When Operations of Distinct Processes are Commutative

Davide Frey, Lucie Guillou, Michel Raynal, François Taïani

Consensus-Free Ledgers
When Operations of Distinct Processes are Commutative

Davide Frey[⋆], Lucie Guillou[⋆], Michel Raynal[⋆,⋄], François Taïani[⋆]

[⋆]Univ Rennes, IRISA, CNRS, Inria, 35042 Rennes, France
[⋄]Department of Computing, Polytechnic University, Hong Kong

Abstract. Considering asynchronous message-passing systems in which any number of processes may crash, this article addresses the construction of ledger objects where (i) the append operations issued from distinct processes commute, while (ii) the append operations issued from the same process do not. In a very interesting way, it appears that the implementation of such ledgers does not need consensus, which makes them both attractive and efficient. Their underlying formalization rests on Mazurkiewicz's traces.

Keywords; Asynchronous system, Blockchain, Commutative operations, Consensus-freedom, FIFO-based synchronization, Immutable memory, Mazurkiewicz's traces, Message-passing, Process crash failures, Reliable broadcast.

1 Introduction

1.1 Context of the study

Once upon a time the Blockchain... Since its introduction, more than ten years ago in the context of cryptocurrencies [13,16], blockchains have receiving more and more attention. A blockchain is nothing more than a technology to implement ledger objects [5,15], i.e., an object providing its users with a list of items (also called blocks, elements, cells, etc. according to the application context) that can be accessed by two operations only, namely an operation append() which allows to add a new element at the head of the list and an operation query() which allows to obtain the current value of the full list. The important point of a ledger lies in the fact that the elements previously added cannot be modified (immutability property).

Basically, and according to the upper layer application, the implementation a ledger object involves two main domains of informatics: synchronization and fault-tolerance (which includes cryptography). The main issue consists then in allowing the processes to agree on the very same order in which items are added to the ledger, i.e., in one way or another, the processes have to solve a consensus problem.

Do all the ledgers need consensus? It has recently been shown that that not all the ledgers need consensus. This actually depends on the application. In an amazing way, it has been show that the synchronization part of cryptocurrencies does not need consensus [2,3,4,6,7]. So, the important point of such *weak* ledgers is immutability. This is the topic addressed in this article in the context of asynchronous message-passing systems where any number of process may commit unexpected crash failures.

1.2 Computing model

Process model The system comprises a set of n sequential asynchronous processes, denoted p_1, ..., p_n. Sequential means that a process invokes one operation at a time, and asynchronous means that each process proceeds at its own speed, which can vary arbitrarily and always remains unknown to the other processes. Any number of processes may crash. A crash is a premature definitive halt. Hence, a process behaves correctly (i.e., executes its algorithm) until it possibly crashes.

From a terminology point of view, when considering an execution, a process that does not crash is *correct*. Otherwise it is *faulty*.

Communication The processes communicate through an underlying message-passing point-to-point network in which there exists a bidirectional channel between any pair of processes. For simplicity, in writing the algorithms, we assume that a process can send messages to itself. Each channel is reliable and asynchronous. Reliable means that a channel neither lose, duplicates, nor corrupts messages. Asynchronous means that the transit delay of each message is finite but arbitrary.

1.3 When the operations append() of different processes commute

Considering the previous computing model, the problem addressed in this article is the following.

- Impose the same view of each local order. Given any process p_i, ensure that all the processes see all the invocations of append() by p_i in the order in which p_i issued them),
- Allow global disorder. This means that the processes may see the appends issued by distinct processes in different order. Let op() denote an append invocation. This means that, if p_i issues op()1 and p_j issues op2(), a process p_k can see first op1() and then op2() while another process sees first op2() and then op1().

One can see that these constraints are the ones required by money transfer: to prevent double spending from occurring, two transfers issued by a process must be seen in their sending order, while transfers from distinct processes may be seen in different order by different processes. Other applications such as work stealing [11] and the distributed simulation of Petri nets belong to the same family of problems. In the context of failure-free systems such an approach based on commutative operations been investigated for about 10 years [9,17].

From an implementation point of view, it is easy to see that, this requires to implement FIFO channels between each pair of processes, which is a simple peer-to-peer problem not requiring consensus. Before presenting a distributed algorithm satisfying the two previous properties, the next section presents a formal definition of the problem based on Mazurkiewicz's traces, which turns to be the theoretical basis on which relies the specification of this family of problems.

2 Underlying Formalization

2.1 A quick look at Mazurkiewicz's traces

A trace monoid (or free partially commutative monoid, also known as Mazurkiewicz's Traces [12,14]) is a generalization of the notion of words (finite sequence over an alphabet Σ, which allow us to capture the independence and the conflicts on operations (represented as letters of Σ).

More formally, a trace monoid over an alphabet Σ is defined by a symmetric independence relation $I \subseteq \Sigma \times \Sigma$ between the letters (operations) of Σ. $(a, b) \in I$ means that the operations a and b commute, i.e. the effect of ab and ba are equivalent.

Two (finite) words $u, v \in \Sigma^*$ are said to be *equivalent under* I, noted $u \overset{I}{\sim} v$, if and only if one can transform u into v (and reciprocally) by exchanging adjacent operations that are independent within u.

Relation $\overset{I}{\sim}$ is an equivalence relation over Σ^*, and a (finite) trace is simply an equivalence class of $\overset{I}{\sim}$, which is a congruence with respect to the concatenation operator (note $\oplus$, but generally omitted), i.e. if $x \overset{I}{\sim} y$ and $u \overset{I}{\sim} v$ then $xu \overset{I}{\sim} yv$. As a result, the concatenation over words translates to the set of traces. More precisely, $[u]_I[v]_I = [uv]_I$, where $u, v \in \Sigma^*$ are words over Σ, $[u]_I$ is the trace represented by u (equivalence class of u under the relation $\overset{I}{\sim}$). The resulting structure $\left(\Sigma^* / \overset{I}{\sim}\right)$ is called *free partially commutative monoid*, denoted $\mathbb{M}(\Sigma, I)$. A subset of $\mathbb{M}(\Sigma, I)$ is called a *trace language*.

2.2 Problem formalization

We consider a ledger with the two types of operations defined below.

- Type A denotes append operations that allow processes to add elements to the ledger. Each append operation returns the symbol $\perp$ (which informs the invoking process it can continue its execution). Let A_i be the bounded set of the append operations invoked only by p_i. Each set A_i is thus attached to a process p_i in the sense that only this process can invoke the operations it contains. The operations in any given A_i do not commute with each other, with respect to the content of the ledger at a given instant, while any operation in A_i and any operation in A_j, with $j \neq i$, commute. In the following, op_i denotes an operation of A_i.
- Type Q denotes query operations that do not modify the ledger and return a value that depends on the current content of the ledger as seen by the invoking process. A query operation can be invoked by any process. In the following, query denotes an operation of Q independently of the process that invokes it.

Process-commutative ledger (PC-ledger) specification Mazurkiewicz's traces allow us to capture the correct behaviors of a ledger. More precisely, a PC-ledger specification is a triple $((A_i)_i, L, Q)$ such that:

- Each set A_i is the set of append operations that p_i can invoke. We define $\Sigma = \bigcup_i A_i$ because the content of the ledger only depends on append operations. Then we leverage the fact that two operations $\mathsf{op}_i \in A_i$ and $\mathsf{op}_j \in A_j$ commute if and only if $i \neq j$ to define an independence relation I over Σ, namely

$$I = (\Sigma \times \Sigma) \setminus \bigcup_{1 \leq i \leq n} (A_i \times A_i).$$

- L is a trace language defined on the monoid $\mathbb{M}(\Sigma, I)$ that satisfies a *forward acceptability* property defined as follows. let t be a trace in $\mathbb{M}(\Sigma, I)$. In the following $\mathsf{mset}(t)$ denotes the multiset of the operations appearing in the trace t. *Forward acceptability* states that for any two traces $u, v \in L$, and any operation $\mathsf{op}_i \in A_i$, we have

  $$u \oplus \mathsf{op}_i \in L \wedge \left(\exists\, \mathsf{op}_k \in \bigcup_{j \neq i} A_j : \mathsf{mset}(v) = \mathsf{mset}(u) \cup \{\mathsf{op}_k\}\right) \Rightarrow v \oplus \mathsf{op}_i \in L.$$

  Forward acceptability means that an append operation op_i issued by a process p_i remains possible ($v \oplus \mathsf{op}_i \in L$) even if a process $p_k \neq p_i$ previously performed an append operation op_k (wherever op_k appears in the trace v).
- The set Q is the set of query operations, each query being a function from the trace language L to a set of application-dependent values. A $\mathsf{query} \in Q$, returns a view of the global content of the ledger as specified by the trace it operates on. Two arbitrary queries issued by two (possibly different) processes will in general return different results, but if the two processes have experienced sequences of operations that correspond to the same trace in L, their queries will return the same value.

The independence relation I expresses the fact that the content of a PC-ledger does not depend on the interleaving of the operations of different processes. Language L, on the other hand, specifies which contents (traces) are valid and through which operations In particular different applications may define L as a more or less constrained subset of $\mathbb{M}(\Sigma, I)$, as long as the forward-acceptability property holds.

Illustration Let us consider money transfer. As previously suggested, this problem can be captured by a PC-ledger specification where the transfer operations by a process p_i define A_i, the invocation of a balance operation is a query, and L is defined as the set of traces of the transfer operations that produce positive balances only. As we can see, the money transfers issued by a process p_i are seen in the same order by all the processes, while money transfers issued by different processes may be seen in different orders. We observe that the corresponding language L satisfies forward acceptability because a transfer operation issued by a process p_i cannot invalidate an outgoing transfer from a different process p_j.

2.3 From a specification to executions

Now that we have defined what is a PC-ledger, we can explain how to make it "live" by defining what is an execution of it on top of an asynchronous crash-prone message-passing system. We do this with the help of the following definitions.

Histories (While different, the following definitions are close to the ones used in [1])

- The *local history* of a process p_i is the sequence E_i of the append and query operations it has executed. If p_i executed $\mathsf{op1}$ before $\mathsf{op2}$ we write $\mathsf{op1} \rightarrow_i \mathsf{op2}$ ($\rightarrow_i$ is called process order).
- A history H is a set of local histories, one per process, $H = (E_1, \cdots, E_n)$.
- Given a history H and a process p_i, let $\widehat{H_i} = (\widehat{E_1}, \cdots, \widehat{E_n})$ such that
  - $\widehat{E_i} = E_i$.
  - $\widehat{E_j} = E_j \setminus Q_j$ for $j \neq i$ where Q_j denotes the set of queries issued by p_j.

Sequential execution A sequential execution SE is a sequence of triplets $SE = (e_x)_x$ where $e_x = (\mathsf{op}, val, i)$, meaning that process p_i invoked $\mathsf{op} \in A_i \cup Q$, with val being the returned value for $\mathsf{op} \in Q$ and $val = \bot$ for $\mathsf{op} \in A_i$.

Let $\mathsf{proj}(SE, \Sigma)$ denote the sequence of append operations in SE, and $[\mathsf{proj}(SE, \Sigma)]_I$ the equivalence class of $\mathsf{proj}(SE, \Sigma)$ under the independence relation $\overset{I}{\sim}$ (defined in Section 2.2).

A sequential execution SE is *legal* if:

- The sequence of append operations is such that $[\mathsf{proj}(SE, \Sigma)]_I \in L$.
- The value returned by a query depends only on the sequence of appends that precede it in SE.

Serializations

- A serialization S of a history H, is a legal sequential execution which contains all operations in H and respects all process orders $(\rightarrow_i)_{1 \leq i \leq n}$.
- Given a history H and a process p_i, a *local* serialization S_i is a serialization of $\widehat{H_i}$.

Distributed PC-ledger object: definition Given a PC-ledger specification $((A_i)_i, L, Q)$, a distributed PC-ledger object is a distributed object whose histories $H = (E_1, \cdots, E_n)$ verify the following properties:

- Any operation invoked by a correct process terminates.
- For any process p_i, there is a local serialization S_i of $\widehat{H_i}$.

3 An Algorithm Implementing a PC-Ledger

3.1 Reliable broadcast

The algorithm that implements a PC-ledger assumes an underlying reliable broadcast communication abstraction. This abstraction provides the processes with two operations denoted $\mathsf{r\_broadcast}()$ and $\mathsf{r\_deliver}()$. When a process invokes $\mathsf{r\_broadcast}(m)$ (resp., $\mathsf{r\_deliver}(m)$), we say it r-broadcasts (resp., r-delivers) the message m. Reliable broadcast is defined by the following properties.

- RB-Validity. If a process p_i r-delivers a message m from a process p_j, then the process p_j r-broadcast m.

– RB-Integrity. Assuming all the messages are different, no process r-delivers twice the same message.
– RB-Termination-1. If a correct process r-broadcasts a message m, it r-delivers it.
– RB-Termination-2. If a correct process r-delivers a message m, all correct processes r-deliver m.

Validity and Integrity are safety properties. Validity relates the outputs to the inputs. Integrity states there is no duplication. The termination properties state that all correct processes r-deliver the same set M of messages, and this set includes all the messages they r-broadcast. Moreover a faulty process r-delivers a subset of M.

Using the technique "first forward and only then deliver", reliable broadcast is easy to implement on top of a point-to-point fully connected network. When a process invokes r_broadcast(m), it sends m to all the processes, and then r-delivers it to itself. When a process receives a message for the first time, it first forwards it to the other processes and only then r-delivers it locally. When a process receives a copy of a message it has already received, it discards it. Algorithms implementing reliable broadcast with additional qualities of service are described in [8,15].

3.2 Local data structures

It is assumed that all the processes know the alphabet Σ (operations) and the language L defining the PC-ledger. The symbol $\oplus$ is used to explicitly denote the concatenation of an element at the end of a sequence. The symbol ϵ denotes the empty sequence. Since L is a trace language, we usually omit the equivalence class notation $[\cdot]_I$ for readability's sake when the context is clear, for instance writing $s \in L$ to mean $[s]_I \in L$ if $s \in \Sigma^*$ is a sequence.

The messages APPLY r-broadcast by the processes contain four fields: the index of the sender process, its sequence number, the identifier of the specific append operation issued by the sender ($opname$), and the parameters of this append operation ($param$).

Each process manages the following local variables.

– sn_i is an sequence number (initialized to 0) used by p_i to identify the messages it r-broadcasts.
– $del_i[1..n]$ is an array of sequence numbers (each initialized to 0. The entry $del_i[j]$ contains the greatest sequence number of the messages p_i has r-delivered from p_j.
– seq_i is the sequence of operations which locally represents the PC-ledger object, as seen by p_i. Its initial value is the empty sequence ϵ.

Algorithm 1 is pretty simple. When a process p_i invokes an operation query(), it locally applies it to its local representation of the PC-ledger seq_i and returns the corresponding result (line 01). By construction, a process p_i only appends operations $\langle opname, param\rangle$ (lines 02-05) that *(i)* belong to the set A_i (the operations it is allowed to use), and *(ii)* are acceptable in p_i's current ledger representation seq_i, namely the concatenation of $\langle opname, param\rangle$ to seq_i remains in the trace language, $[seq_i \oplus \langle opname, param\rangle]_I \in L$. When it invokes append$(opname, param)$, with $\langle opname, param\rangle \in A_i$, p_i first increases sn_i and r-broadcasts the message APPLY$(opname, param, sn_i, i)$ to all the processes (including itself, lines 02-03).

init: $sn_i \leftarrow 0$; $seq_i \leftarrow \emptyset$; $del_i[1..n] \leftarrow [0, \cdots, 0]$.

operation $\mathsf{query}()$ **is** % $\mathsf{query}()$ is any operation of type Q
(01) $res \leftarrow \mathsf{query}(seq_i)$; $\mathsf{return}(res)$.

operation $\mathsf{append}(opname, param)$ **is** % $\langle opname, param \rangle$ is any operation $\in A_i$
(02) $sn_i \leftarrow sn_i + 1$;
(03) r_broadcast $\text{APPLY}(opname, param, sn_i, i)$;
(04) wait $(del_i[i] = sn_i)$;
(05) $\mathsf{return}()$.

when $\text{APPLY}(opname, param, sn, j)$ **is** r_delivered **do**
(06) $\mathsf{wait}(sn = del_i[j] + 1) \wedge (seq_i \oplus \langle opname, param \rangle \in L)$;
(07) $seq_i \leftarrow seq_i \oplus \langle opname, param \rangle$;
(08) $del_i[j] \leftarrow del_i[j] + 1$.

Algorithm 1: An algorithm implementing a PC-ledger (code for p_i)

When p_i r-delivers a message $\text{APPLY}(opname, param, sn, j)$, it waits (line 06) until it has processed the previous append from p_j, and this new append satisfies the forward acceptability property. When this occurs, p_i adds this append to seq_i (line 07) and accordingly updates $del_i[j]$ (line 08).

4 Proof of the Algorithm

Notation Considering an invocation op_j of $\mathsf{append}()$ by a process p_j, we note $\mathsf{before}(\mathsf{op}_j)$ all $\mathsf{append}()$ invocations by p_j that precede op_j.

Lemma 1. *Any invocation of an operation by a process that does not crash terminates.*

Proof Let us first observe that any invocation of an operation $\mathsf{query}()$ by a correct process trivially terminates.

As far a the operation $\mathsf{append}()$ is concerned, let us assume (by contradiction) that some invocation op_j of an append invocation issued by a correct process p_j never returns. Since p_j is correct, by RB-Termination-1 p_j eventually r-delivers the message m_{op_j} corresponding to op_j. Let sn be the sequence number associated with op_j. Let us observe that all the append invocations issued by p_j with a sequence number smaller than sn have terminated (otherwise p_j could not have issued an operation with sequence number sn) and have therefore been processed at lines 07 and 08. It follows that the predicate $sn = del_j[j] + 1$ (line 06) is satisfied when m_{op_j} is r-delivered.

Let us note seq_j^0 the value of seq_j when op_j is invoked by p_j, and seq_j^1 its value when m_{op_j} is r-delivered by p_j. By assumption, we have $seq_j^0 \oplus \mathsf{op}_j \in L$, since no nodes is Byzantine (for the sake of conciseness, we equate here op_j with its associated $\langle opname, param \rangle$ pair). Because all these invocations have already been processed by p_j when op_j is invoked, we have $\mathsf{before}(\mathsf{op}_j) \subseteq \mathsf{mset}(seq_j^0)$. Because p_j does not perform any additional $\mathsf{append}()$ invocation after op_j (since by assumption op_j never returns), we also have $\mathsf{mset}(seq_j^1) = \mathsf{mset}(seq_j^0) \cup A'$ for some $A' \subseteq \Sigma$ that fulfills $A' \cap A_j = \emptyset$. By recursively applying the *forward acceptability* property (defined in Section 2.2), this implies that $seq_j^1 \oplus \mathsf{op}_j \in L$, and therefore that the second predicate at line 06 is also satisfied when m_{op_j} is r-delivered, leading to the execution of line 08, and the termination of $\mathsf{append}()$ at line 04. $\Box_{Lemma\ 1}$

Notations

- Let op_j^{sn} denote the append operation issued by p_j with sequence number sn. Hence the message $\text{APPLY}(opname, param, sn, j)$ is associated with this operation.
- A process p_i locally *processes* the operation op_j^{sn} when, after it r-delivered the message $\text{APPLY}(opname, param, sn, j)$, it executes the lines 07-08).
- If a message $\text{APPLY}(opname, param, sn, j)$ be is r-delivered by a correct process, we say it is *successful*. It follows from the RB-Termination properties that all the operations $\mathsf{append}()$ invoked by the correct processes give rise to successful APPLY messages.

Lemma 2. *If a process p_i processes op_j^k, any correct process processes it.*

Proof Let us assume that op_j^k is the s^{th} operation processed by p_i. We prove the lemma by induction on s. Let us note seq_i^{op} and del_i^{op} the values of seq_i and del_i at line 06 when the wait statement becomes true for op_j^k at p_i, and op_j^k is selected by p_i to be added to its ledger.

For $s = 1$, seq_i^{op} is the empty sequence ϵ, and $del_i^{\mathsf{op}}[j] = 0$ (since p_i has not processed any operation yet from any process). As the wait statement has just become true, we therefore have $k = del_i^{\mathsf{op}}[j] + 1 = 1$, and $seq_i^{\mathsf{op}} \oplus \mathsf{op}_j^k = \epsilon \oplus \mathsf{op}_j^k = \mathsf{op}_j^k \in L$ (since p_i is not Byzantine).

Let us consider a correct process p_ℓ. Due to the RB-Termination-2 property, p_ℓ eventually r-delivers the message $m_j^k = \text{APPLY}(opname, param, k, j)$ from p_j associated with op_j^k. Let us write seq_ℓ^{op} and del_ℓ^{op} the values of seq_ℓ and del_ℓ at line 06 just after m_j^k has been r-delivered.

By RB-Integrity, this is the first (and only) time p_ℓ r-delivers m_j^k, which implies $del_\ell[j]$ has not yet taken the value $k = 1$, and by monotony that $del_\ell^{\mathsf{op}}[j] = 0$. $del_\ell^{\mathsf{op}}[j] = 0$ implies that p_ℓ has not processed any operation from p_j yet, and therefore that $\mathsf{mset}(seq_\ell^{\mathsf{op}}) \subseteq \bigcup_{k' \neq j} A_k$. This last inclusion and the fact that $\mathsf{op}_j^k \in L$ (see above) implies by recursively using the *forward acceptability* property (Section 2.2) that $seq_\ell \oplus \mathsf{op}_j^k \in L$, and with $k = 1 = del_\ell^{\mathsf{op}}[j] + 1$ that the wait statement is immediately verified by p_ℓ, and that op_j^k is processed by p_ℓ, concluding the proof for $s = 1$.

Let us now assume that the property is true up to a value $s - 1 > 0$. When the wait statement becomes true for op_j^k at p_i, we have $del_i^{\mathsf{op}}[j] = k - 1$, implying p_i has already processed all the earlier operations $\mathsf{before}(\mathsf{op}_j^k) = \{\mathsf{op}_j^{k'}\}_{k' < k}$ issued by p_j, but has not yet processed any additional operation from p_j. As a consequence $\mathsf{mset}(seq_i^{\mathsf{op}}) \cap A_j = \mathsf{before}(\mathsf{op}_j^k)$. Furthermore, as earlier, we also have $seq_i^{\mathsf{op}} \oplus \mathsf{op}_j^k \in L$.

Let us consider a correct process p_ℓ. As earlier, p_ℓ eventually r-delivers the message m_j^k. Let us assume the condition of the wait statement at line 06 never becomes true for

op_j^k, and op_j^k is never processed by p_ℓ. By induction hypothesis, since p_i has already processed all the operations in $\mathsf{mset}(seq_i^{\mathsf{op}})$, and $|\mathsf{mset}(seq_i^{\mathsf{op}})| = s - 1$, we know that p_ℓ also eventually processes all the operations in $\mathsf{mset}(seq_i^{\mathsf{op}})$, and at some point we have $\mathsf{mset}(seq_i^{\mathsf{op}}) \subseteq \mathsf{mset}(seq_\ell)$, and therefore $\mathsf{before}(\mathsf{op}_j^k) \subseteq \mathsf{mset}(seq_\ell)$, and $del_\ell[j] \geq k - 1$. Furthermore, since p_ℓ never processes op_j^k, we also have $del_\ell[j] < k$, and hence $del_\ell[j] = k - 1$, which implies $\mathsf{mset}(seq_\ell^{\mathsf{op}}) \cap A_j = \mathsf{before}(\mathsf{op}_j^k)$. As a result, there exists some set $A' \subseteq \bigcup_{k' \neq j} A_k$, such that $\mathsf{mset}(seq_\ell) = \mathsf{mset}(seq_i^{\mathsf{op}}) \cup A'$. Using $seq_i^{\mathsf{op}} \oplus \mathsf{op}_j^k \in L$, we can recursively apply the *forward acceptability* property, leading to $seq_\ell \oplus \mathsf{op}_j^k \in L$, meaning the wait statement eventually becomes true for op_j^k at p_ℓ, which contradicts the fact it is never processes by p_ℓ, and concludes the proof.

$\square_{Lemma\ 2}$

Theorem 1. *Algorithm* 1 *implements a PC-ledger.*

Proof The fact that the operations issued by the correct processes terminate follows from Lemma 1. So, the rest of the proof concerns the safety properties of a PC-ledger, namely: for any process p_i, there is a serialization S_i of $\widehat{H_i}$ (i.e. a legal sequential execution that is equivalent to $\widehat{H_i}$ from p_i's viewpoint).

Considering a process p_i, let us first recall the definition of $\widehat{H_i}$, namely $\widehat{H_i} = (\widehat{E_1}, \cdots, \widehat{E_n})$ such that $\widehat{E_i}$ is the local history of p_i, and, for each $j \neq i$, $\widehat{E_j}$ is the local history of p_j including only its append operations.

From Lemma 2 and the fact that the messages APPLY() are processed in their sending order (from a local point-to-point point of view) and in agreement with the forward acceptability property (from a global point of view), it follows that the append operations issued by any process p_j are added to seq_i in the order they have been invoked by p_j. Moreover, the queries issued by p_i are on the value of seq_i at the query time. It follows that the corresponding sequence of append issued by the processes and the query operations issued by p_i is a serialization S_i of $\widehat{H_i} = (\widehat{E_1}, ..., \widehat{E_n})$. Moreover, as the value returned by each query issued by p_i depends on all the append operations that precede it in S_i, the serialization is legal. $\square_{Theorem\ 1}$

5 Conclusion

Considering asynchronous message-passing systems in which any number of processes may commit crash failures, this article has introduced the notion of a ledger where append operations from distinct processes are commutative while operations from a same processes are not (hence the name *PC-ledger* where PC stands for Process-Commutative).

After the formal definition of such objects, the article has shown how these objects can be implemented in asynchronous crash-prone distributed systems. On an application point of view, as already noticed, it is interesting to notice that, while money transfers from a given user are not commutative (in order to prevent double spending from occurring), money transfers from different users are commutative, and consequently money-transfer ledgers belong to the family of PC-ledgers.

Interestingly enough, this article has also shown the study of the class of applications where, while the operations of each process are not commutative, the operations issued by distinct processes are, can be based on sane foundations, namely Mazurkiewicz's traces.

This paper has shown that the *trace languages*-based approach allows us to cope with the net effect produced by adversaries such as asynchrony and process crashes. So, and last but not least, a far from being trivial problem concerns the adversarial context defined by asynchrony and Byzantine process failures [10]. Can the proposed trace languages-based approach be used to address such a stronger non-deterministic context?

Acknowledgments

This work was partially supported by the the French ANR project ByBLoS (ANR-20-CE25-0002-01) devoted the modular design of building blocks for large-scale fault-tolerant multi-users applications.

References

1. Ahamad M., Neiger G., Burns J.E., Hutto P.W., and Kohli P., Causal memory: definitions, implementation and programming. *Distributed Computing*, 9:37–49 (1995)
2. Auvolat A., Frey D., Raynal M., and Taïani F., Money transfer made simple. *Bulletin of the European Association of Theoretical Computer Science (EATCS)*, Vol. 132, pp. 22-43 (2020)
3. Cholvi V., Fernández Anta A., Georgiou Ch., Nicolaou N.C., Raynal M., and Russo A., Byzantine-tolerant distributed grow-only sets: specification and applications. *Submitted*.
4. Collins D., Guerraoui R., Komatovic J., Kuznetsov P., Monti M., Pavlovic M., Pignolet Y.A., Seredinschi, D., Tonkikh A., and Xygkis A., Online payments by merely broadcasting messages. *Proc. 50th IEEE/IFIP Int'l Conference on Dependable Systems (DSN'20)*, IEEE Press, pp. 26-38 (2020)
5. Fernández Anta A., Konwar M.K., Georgiou Ch., and Nicolaou N.C., Formalizing and implementing distributed ledger objects. *SIGACT News*, 49(2):58-76 (2018)
6. Guerraoui R., Kuznetsov P., Monti M. ,Pavlovic M., Seredinschi D.A., The consensus number of a cryptocurrency. *Proc. 38th ACM Symposium on Principles of Distributed Computing (PODC'19)*, ACM Press, pp. 307–316 (2019)
7. Gupta S., A non-consensus based decentralized financial transaction processing model with support for efficient auditing. *Master Thesis*, Arizona State University, 83 pages (2016)
8. Hadzilacos V. and Toueg S., A modular approach to fault-tolerant broadcasts and related problems. *Tech Report 94-1425*, 83 pages, Cornell University (1994)
9. Li Ch., Porto D., Clement A., Gehrke J., Preguiça N.M., and Rodrigues R., Making geo-replicated systems fast as possible, consistent when necessary. *Proc. 10th Symposium on Operating Systems Design and Implementation (OSDI'12)*, USENIX Association , pp. 265-278 (2012)
10. Lamport L., Shostack R., and Pease M., The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3)-382-401 (1982)
11. Michael M.M., Vechev M.T., and Saraswat S.A., Idempotent work stealing. *Proc. 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP'09)*, ACM Press, pp. 45–54 (2009)

12. Mazurkiewicz A.W., Trace theory. *Petri Nets : Applications and Relationships to Other Models of Concurrency*, Springer LNCS 255, pp. 279-324 (1986)
13. Nakamoto S., Bitcoin: a peer-to-peer electronic cash system. https://bitcoin.org/bitcoin.pdf (2008)
14. Petit A., Recognizable trace languages, distributed automata and the distribution problem. *Acta Informatica*, 30(1):89-101 (1993)
15. Raynal M., *Fault-tolerant message-passing distributed systems: an algorithmic approach. Springer*, 550 pages, ISBN: 978-3-319-94140-0 (2018)
16. Riesen A., Satoshi Nakamoto and the financial crisis of 2008. https://andrewriesen.me/2017/12/18/2017-12-18-satoshi-nakamoto-and-the-financial-crisis-of-2008/ [last accessed April 22, 2020]
17. Shapiro M., Preguiça N., Baquero C., and Zawirski M., Conflict-free replicated data types. *Proc. 13th Int'l Symp. on Stabilization, Safety, and Security of Distributed Systems (SSS'11)*, Springer LNCS 6976, pp. 386-400 (2011)

A Exercise: From a PC-Ledger to a Distributed PC-State Machine

Some applications do not require to keep the full history saved in a ledger object. In this case, it is easy to replace the sequence seq_i by a local variable $state_i$ which represents the current state of the ledger as know by p_i. The resulting PC-state machine Algorithm 2 is trivially obtained from Algorithm 1. The function $\delta()$ is the transition function of the corresponding state machine, which is assumed to return $\perp$ in case a transition is not allowed.

init: $sn_i \leftarrow 0$; $state_i \leftarrow$ initial value of the state machine; $del_i[1..n] \leftarrow [0, \cdots, 0]$.

operation query() **is** % query() is any operation of type Q
$res \leftarrow$ query($state_i$); return(res).

operation append($opname, param$) **is** % $\langle opname, param \rangle$ is any operation $\in A_i$
$sn_i \leftarrow sn_i + 1$;
r_broadcast APPLY($opname, param, sn_i, i$);
wait ($del_i[i] = sn_i$);
return().

when APPLY($opname, param, sn, j$) **is** r_delivered **do**
wait($sn = del_i[j] + 1) \wedge (\delta(seq_i, \langle opname, param \rangle) \neq \perp$);
$state_i \leftarrow \delta(seq_i, \langle opname, param \rangle)$;
$del_i[j] \leftarrow del_i[j] + 1$.

Algorithm 2: From a PC-ledger to a PC-state machine (code for p_i)