Why You Can't Beat Blockchains

Consistency and High Availability in Distributed Systems

Alain Girault[1], Gregor Gössler[1], Rachid Guerraoui[2],
Jad Hamza[3], and Dragos-Adrian Seredinschi[2]

[1] Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG, 38000 Grenoble, France
[2] LPD, EPFL
[3] LARA, EPFL

Abstract. We study the issue of data consistency in highly-available distributed systems. Specifically, we consider a distributed system that replicates its data at multiple sites, which is prone to partitions, and which is expected to be highly available. In such a setting, strong consistency, where all replicas of the system apply synchronously every operation, is not possible to implement. However, many weaker consistency criteria that allow a greater number of behaviors than strong consistency, are implementable in distributed systems.
We focus on determining the strongest consistency criterion that can be implemented in a distributed system that tolerates partitions. We show that no criterion stronger than Monotonic Prefix Consistency (MPC) can be implemented. MPC is the consistency criterion underlying blockchains.

1 Introduction

Replication is a mechanism that enables sites from different geographical locations to access a shared data structure with low latency. It consists of creating copies of this data structure on each *site* of a distributed system. Ideally, replication should be transparent, in the sense that the users of the data structure should not notice discrepancies between the different copies of the data structure.

An ideal replication scheme could be implemented by keeping all sites synchronized after each update to the data structure. This ideal model is called *strong consistency*, or linearizability [1]. The disadvantage of this model is that it can cause large delays for users, and worse the data structure might not be *available* to use at all times. This may happen, for instance, if some sites of the system are unreachable, i.e., partitioned from the rest of the network. Briefly, it is not possible to implement strong consistency in a distributed system while ensuring high availability [2,3].

Given this impossibility, developers rely on weaker notions of consistency, such as causal consistency [4]. Weaker consistency criteria do not require sites to be exactly synchronized as in strong consistency. For instance, causal consistency allows different sites to apply updates to the data structure in different orders, as long as the updates are not *causally related*. Informally, a consistency criterion specifies the behaviors that are allowed by a replicated data structure. In this sense, causal consistency is more permissive than strong consistency. We also say that strong consistency is *stronger* than causal consistency, as strong consistency allows strictly fewer behaviors than causal consistency. A natural question is then: What is the strongest consistency criterion that can be implemented by a replicated data structure?

In [5], it was proven that nothing stronger than *observable causal consistency* (a variant of causal consistency) can be implemented. It is an open question whether observable causal consistency itself is actually implementable. Moreover, [5] does not study consistency criteria that are not comparable to observable causal consistency. Indeed, there exist consistency criteria that are neither stronger than causal consistency, nor weaker, and which can be implemented by a replicated data structure.

In our paper, we explore one such consistency criterion. More precisely, we prove that, under some conditions which are natural in a large distributed system, nothing stronger than *monotonic prefix consistency* (`MPC`) [6] can be implemented. This result does not contradict the result from [5], since `MPC` and causal consistency are incomparable.

The reason why `MPC` and observable causal consistency are incomparable is as follows. `MPC` requires all sites to apply updates in the same order (but not necessarily synchronized at the same time, as in strong consistency), while causal consistency allows non-causally related updates to be applied in different orders. On the other hand, causal consistency requires all causally-related updates to be applied in an order respecting causality, while `MPC` requires no such constraint.

`MPC` corresponds to the consistency criterion that *blockchains* implement with high probability [7,8,9]. A blockchain is a replicated data structure, composed of a list of blocks. Under some conditions, blocks can be appended at the end of the list, and all participants of the blockchain agree on the order in which blocks are appended.

Overall, our contribution is to prove that, for a notion of behavior where updates are anonymous and their times and places of origin do not matter (as is the case in large-scale open implementations such as blockchains), nothing stronger than `MPC` can be implemented in a distributed setting. Blockchains therefore implement a consistency model which is closest to strong consistency and is achievable in a distributed setting. Moreover, we remark that, clients who are only sensitive to our notion of behavior cannot tell the difference between a strongly consistent implementation and an `MPC` implementation.

In the rest of this paper, we first give preliminary notions and a formal definition of the problem we're addressing (sections 2 and 3). We then turn our attention to the `MPC` model by defining it formally and through an implementation (Section 4). We prove that, given the notion of behavior mentioned above, and under conditions natural in a large-scale network (availability, convergence), nothing stronger than `MPC` can be implemented (Section 5). Then we compare `MPC` with other consistency models (Section 6), and conclude (Section 7).

2 Implementations of Replicated Data Structures

An *implementation* of a replicated data structure consists of several *sites* that communicate by sending messages. Messages are delivered asynchronously by the network, and can be reordered or delayed. To be able to build implementations that provide liveness guarantees, we assume all messages are *eventually* delivered by the network.

Each site of an implementation maintains a local state. This local state reflects the view that the site has on the replicated data structure, and may contain arbitrary data. Each site implements the protocol by means of an *update handler*, a *query handler*, and a *message handler*.

The update handler is used by clients to submit updates to the data structure. The update handler may modify the local states of the site, and broadcast a message to the other sites. Later, when another site receives the message, its *message handler* is triggered, possibly updating the local state of the site, and possibly broadcasting a new message.

The *query handler* is used by clients to make queries on the data structure. The query handler returns an answer to the client, and is a read-only operation that does not modify the local state or broadcast messages.

Remark 1. Our model only supports broadcast and not general peer-to-peer communication, but this is without loss of generality. We can simulate sending a message to a particular site by writing the identifier of the receiving site in the broadcast message. All other sites would then simply ignore messages that are not addressed to them.

In this paper, we consider implementations of the *list data structure*. The list supports an update operation of the form $\texttt{write}(d)$, with $d \in \mathbb{N}$, which adds the element d to the list. The list also supports a query operation $\texttt{read}$ that returns the whole list $\ell \in \mathbb{N}^*$, which is a sequence of elements in $\mathbb{N}$.

Definition 1. *Let* $\mathsf{Upd} = \{\texttt{write}(d) \mid d \in \mathbb{N}\}$ *be the set of updates, and* $\mathsf{Ans} = \{\texttt{read}(\ell) \mid \ell \in \mathbb{N}^*\}$ *be the set of all possibles answers to queries.*

We focus on the list data structure because queries return the history of all updates that ever happened. In that regard, lists can encode any other data structure whose operations can be split in updates and queries, by adding a processing layer after the query operation of the list returns all updates. Data structures that contain operations which are queries and updates at the same time (e.g. the Pop operation of a stack) are outside the scope of this paper. We now proceed to give the formal syntax for implementations, and then the corresponding operational semantics.

Definition 2. *An* implementation $\mathcal{I}$ *is a tuple* $(Q, \iota, \mathbb{P}, \mathsf{Msg}, \mathsf{msg\_handler}, \mathsf{update\_handler}, \mathsf{query\_handler})$ *where*

- Q *is a non-empty set of* local states*,*
- $\mathbb{P}$ *is a non-empty finite set of* process identifiers*,*

- $\iota : \mathbb{P} \to Q$ *associates to each process an* initial local state*,*
- Msg *is a set of* messages*,*
- $\mathsf{msg\_handler} : Q \times \mathsf{Msg} \to Q \times \mathsf{Msg}^{\perp}$ *is a function, called the* handler of incoming messages*, which updates the local state of a site when a message is received, and possibly broadcasts a new message,*
- $\mathsf{update\_handler} : Q \times \mathsf{Upd} \to Q \times \mathsf{Msg}^{\perp}$ *is a function, called the* handler of updates*, which modifies the local state when an update is submitted, and possibly broadcasts a message.*
- $\mathsf{query\_handler} : Q \to \mathsf{Ans}$ *is a function, called the* handler of client queries*, which returns an answer to client queries.*

The set $\mathsf{Msg}^{\perp}$ *is defined as* $\mathsf{Msg} \uplus \{\perp\}$*, where* $\perp$ *is a special symbol denoting the fact that no message is sent.*

Before defining the semantics of implementations, we introduce a few notations. We first define the notion of *action*, used to denote events that happen during the execution. Each action contains a unique *action identifier aid* $\in \mathbb{N}$, and the process identifier $pid \in \mathbb{P}$ where the action occurs.

Definition 3. *A* broadcast action *is a tuple* $(aid, pid, \texttt{broadcast}(mid, msg))$*, and a* receive action *is a tuple* $(aid, pid, \texttt{receive}(mid, msg))$*, where* $mid \in \mathbb{N}$ *is the* message identifier *and* $msg \in \mathsf{Msg}$ *is the message. An* update action *or a* write action *is a tuple* $(aid, pid, \texttt{write}(d))$ *where* $d \in \mathbb{N}$*. Finally, a* query action *or a* read action *is a tuple* $(aid, pid, \texttt{read}(\ell))$ *where* $\ell \in \mathbb{N}^*$*.*

Executions are then defined as sequences of actions, and are considered up to action and message identifiers renaming.

Definition 4. *An* execution *e is a sequence of broadcast, receive, query and update actions where no two actions have the same identifier aid, and no two broadcast actions have the same message identifier mid.*

We now describe how implementations operate on a given site $pid \in \mathbb{P}$.

Definition 5. *We say that a sequence of actions* $\sigma_1 \dots \sigma_n \dots$ *from site pid* follows $\mathcal{I}$ *if there exists a sequence of states* $q_0 \dots q_n \dots$ *such that* $q_0 = \iota(pid)$*, and for all* $i \in \mathbb{N}\backslash\{0\}$*, we have:*

1. *if* $\sigma_i = (aid, pid, \texttt{write}(d))$ *with* $d \in \mathbb{N}$*, then* $\mathsf{update\_handler}(q_{i-1}, \texttt{write}(d)) = (q_i, \_)$*. This means that upon a write action, a site must update its state as defined by the update handler;*
2. *if* $\sigma_i = (aid, pid, \texttt{read}(\ell))$ *with* $\ell \in \mathbb{N}^*$*, then* $\mathsf{query\_handler}(q_{i-1}) = \texttt{read}(\ell)$ *and* $q_i = q_{i-1}$*. This condition states that query actions do not modify the state, and that the answer* $\texttt{read}(\ell)$ *given to query actions must be as specified by the query handler, depending on the current state* q_{i-1}*;*
3. *if* $\sigma_i = (aid, pid, \texttt{broadcast}(mid, msg))$*, then* $q_i = q_{i-1}$*. Broadcast actions do not modify the local state;*
4. *if* $\sigma_i = (aid, pid, \texttt{receive}(mid, msg))$*, then* $\mathsf{msg\_handler}(q_{i-1}, msg) = (q_i, \_)$*. The reception of a message modifies the local state as specified by* $\mathsf{msg\_handler}$*.*

Moreover, we require that broadcast actions are performed if and only if they are triggered by the handler of incoming messages, or the handler of clients requests. Formally, for all $i > 0$*,* $\sigma_i = (aid, pid, \texttt{broadcast}(mid, msg))$ *if and only if either:*

5. $\exists\, \texttt{write}(d) \in \mathsf{Upd}$ *and* $aid' \in \mathbb{N}$ *such that* $\sigma_{i-1} = (aid', pid, \texttt{write}(d))$ *and* $\mathsf{update\_handler}(q_{i-1}, \texttt{write}(d)) = (q_i, msg)$*, or*
6. $\exists\, aid' \in \mathbb{N}$*,* $mid \in \mathbb{N}$*, and* $msg' \in \mathsf{Msg}$ *such that* $\sigma_{i-1} = (aid', pid, \texttt{receive}(mid, msg))$ *and* $\mathsf{msg\_handler}(q_{i-1}, msg') = (q_i, msg)$.

When all conditions hold, we say that $q_0 \dots q_n \dots$ *is a* run *for* $\sigma_1 \dots \sigma_n \dots$. *Note that when a run exists for a sequence of actions, it is unique.*

We then define the set of executions generated by $\mathcal{I}$, denoted $[\![\mathcal{I}]\!]$. In particular, this definition models the communication between sites, and specifies that a receive action may happen only if there exists a broadcast action with the same message identifier preceding the receive action in the execution. Moreover, a *fairness* condition ensures that, in an infinite execution, every broadcast action must have a corresponding receive action on every site.

Definition 6. *Let* $\mathcal{I}$ *be an implementation. The set of executions generated by* $\mathcal{I}$ *is* $[\![\mathcal{I}]\!]$ *such that* $e \in [\![\mathcal{I}]\!]$ *if and only if the three following conditions hold:*

– ***Projection:*** *for all* $pid \in \mathbb{P}$*, the projection* $e|_{pid}$ *follows* $\mathcal{I}$*,*
– ***Causality:*** *for every receive action* $\sigma = (aid, pid, \texttt{receive}(mid, msg))$*, there exists a broadcast action* $(aid', pid', \texttt{broadcast}(mid, msg))$ *before* σ *in* e*,*
– ***Fairness:*** *if* e *is infinite, then for every site* $pid \in Pid$ *and every broadcast action* $(aid', pid', \texttt{broadcast}(mid, msg))$ *performed on any site* pid'*, there exists a receive action* $(aid, pid, \texttt{receive}(mid, msg))$ *in* e*,*

where $e|_{pid}$ *is the subsequence of* e *of actions performed by process pid:*

– $\varepsilon|_{pid} = \varepsilon$*;*
– $((aid, pid, x).e)|_{pid} = (aid, pid, x).(e|_{pid})$*;*
– $((aid, pid', x).e)|_{pid} = e|_{pid}$ *whenever* $pid' \neq pid$*.*

For the rest of the paper, we consider that updates are unique, in the sense that an execution may not contain two update actions that write the same value $d \in \mathbb{N}$. This assumption only serves to simplify the presentation of our result, and can be done without loss of generality. In practice, updates can be made unique by attaching a unique timestamp to them.

3 Problem Definition

In this section, we explain how we compare implementations using the notion of *trace*. Informally, the trace of an execution corresponds to what is observable from the point of view of clients using the data structure.

Our notion of trace is based on two assumptions: 1) Clients know the order of the queries they have done on a site, but not the relative positions of their queries with respect to other clients' queries, 2) Updates are anonymous, and their origin is not relevant for the implementation. This models freely available data structures, such as the Bitcoin blockchain, where any person can disseminate a transaction in the network, and the place and time where the transaction was created are not relevant for the protocol execution.

More precisely, a trace records an unordered set of *anonymous* updates (without a site identifier), and records for each site the sequence of queries that happened on this site.

Definition 7. *The* trace (t_r, W) *corresponding to an execution* e *is denoted* $\mathsf{tr}(e)$*, where* $t_r = (A, <, \mathit{label})$ *is a labelled partially ordered set such that:*

- A *is the set of action identifiers of query actions of* e*;*
- $<$ *is a transitive and irreflexive relation over* A*, sometimes called the* program order*, ordering queries performed on the same site; more precisely, we have* $aid < aid'$ *if* $aid, aid' \in A$ *are action identifiers performed by the same site, and that appear in that order in* e*;*
- $\mathit{label} : A \to \mathsf{Ans}$ *is the* labelling function *such that for any* $aid \in A$*,* $\mathit{label}(aid)$ *is the answer of the query action corresponding to* aid *in* e*;*

and $W \subseteq \mathbb{N}$ *is the set of elements that appear in an update action of* e*.*

We illustrate this definition with the following example.

Example 1. Consider the execution e in Figure 1, and its corresponding trace $\mathsf{tr}(e)$. ($pid_1, pid_2, pid_3 \in \mathbb{P}$ are site identifiers, $mid_1, mid_2, mid_3 \in \mathbb{N}$ are unique message identifiers, and $msg_1, msg_2, msg_3 \in \mathsf{Msg}$ are messages.)

Then, we compare implementations by looking at the set of traces they produce. The fewer traces an implementation produces, the stronger it is, and the closer it is to strong consistency.

Definition 8. *The notation* $\mathsf{tr}()$ *is extended to sets of executions point-wise. An implementation* $\mathcal{I}_1$ *is* stronger *than* $\mathcal{I}_2$*, denoted* $\mathcal{I}_1 \preceq \mathcal{I}_2$ *iff*

$$\mathsf{tr}([\![\mathcal{I}_1]\!]) \subseteq \mathsf{tr}([\![\mathcal{I}_2]\!])$$

The implementations $\mathcal{I}_1$ *and* $\mathcal{I}_2$ *are said to be* equivalent*, denoted* $\mathcal{I}_1 \approx \mathcal{I}_2$*, iff* $\mathcal{I}_1 \preceq \mathcal{I}_2$ *and* $\mathcal{I}_2 \preceq \mathcal{I}_1$*. Moreover,* $\mathcal{I}_1$ *is* strictly stronger *than* $\mathcal{I}_2$*, denoted* $\mathcal{I}_1 \prec \mathcal{I}_2$*, iff* $\mathcal{I}_1 \preceq \mathcal{I}_2$ *and* $\mathcal{I}_1 \not\approx \mathcal{I}_2$*.*

Our goal is to identify the strongest implementations. These are the implementations that are minimal according to the order $\preceq$. More specifically, these are the implementations $\mathcal{I}$ for which there does not exist an implementation $\mathcal{I}'$ strictly stronger than $\mathcal{I}$.

$(188, pid_1, \texttt{write}(3))\cdot$
$(3713, pid_1, \texttt{broadcast}(mid_3, msg_3))\cdot$
$(152, pid_1, \texttt{write}(1))\cdot$
$(16, pid_1, \texttt{broadcast}(mid_1, msg_1))\cdot$
$(137, pid_1, \texttt{read}[])\cdot$
$(2448, pid_3, \texttt{read}[])\cdot$
$(37, pid_2, \texttt{write}(2))\cdot$
$(164, pid_2, \texttt{broadcast}(mid_2, msg_2))\cdot$
$(189, pid_2, \texttt{read}[2])\cdot$
$(733, pid_3, \texttt{receive}(mid_2, msg_2))\cdot$
$(133, pid_3, \texttt{receive}(mid_1, msg_1))\cdot$
$(111, pid_2, \texttt{receive}(mid_1, msg_1))\cdot$
$(17, pid_3, \texttt{read}[2, 1])\cdot$
$(13, pid_1, \texttt{receive}(mid_2, msg_2))\cdot$
$(12, pid_1, \texttt{read}[2, 1])\cdot$
$(15, pid_2, \texttt{read}[2, 1])$

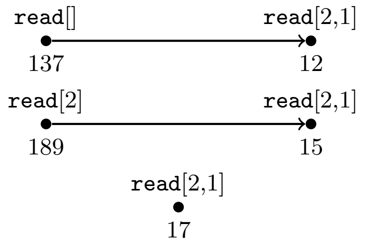

Fig. 1: An execution e (top) and its corresponding trace $\mathsf{tr}(e) = (t_r, W)$ (bottom). The dots represent the action identifiers of t_r (written under the dot), and the corresponding labels are represented right above. The arrows represent the program order $<$ of t_r. The set W is {1,2,3}.

4 Monotonic Prefix Consistency (MPC)

4.1 Description of MPC

Often called consistent prefix [6,10], the MPC model requires that all sites of the replicated system agree on the order of write operations (i.e., updates on the state). There exists a global order such that every read operation, whatever the site that performs it, always returns a prefix of this order. Moreover, read operations which execute on the same site are monotonic. This means that subsequent reads at the same site reflect a non-decreasing prefix of writes, i.e., the prefix must either increase or remain unchanged.

Note that the global order on write operations on which the sites agree does not necessarily satisfy causality among these operations nor real-time. In other words, the order in which clients submit write operations does not translate into any constraints on the global order in which these updates apply at all sites. With respect to freshness, MPC does not guarantee that a read operation will return *all* of the preceding writes, only a prefix of these writes. For instance, some sites can be later than other sites in applying some updates.

```
// Each site stores an element of Q, defined as a list of numbers
type Q = List[Nat]


// The implementation makes use of two kinds of messages
abstract class Msg
// Forwarded messages go from Site i to Site 1, for all i > 1
case class Forwarded(d: Nat) extends Msg
// Apply messages originate from Site 1 and go to Site i, for i > 1
case class Apply(d: Nat) extends Msg


// The update handler for Site 1 appends element ‘upd’ to the
// list, and sends an Apply(upd) message to the other sites, telling
// them to do the same
def update_handler(q: Q, upd: Upd) = (append(q,upd), Apply(upd))

// The update handler for Site i > 1 sends a message Forwarded(upd)
// which is destined for Site 1, and does not modify the state
def update_handler(q: Q, upd: Upd) = (q, Forwarded(upd))


// Message handler for Site 1 (ignores Apply messages)
def msg_handler(msg: Msg) = msg match {
  case Forwarded(d) => (append(q,d), Apply(upd))
}

// Message handler for Site i > 1 (ignores Forwarded messages)
def msg_handler(msg: Msg) = msg match {
  case Apply(d) => (append(q,d), ⊥)
}


// The query handler of any site returns the current local
// state on that site
def query_handler(q: Q) = q
```

Fig. 2: An implementation of `MPC`. For ease of presentation, we assume here that update and message handlers can be different depending on the site. This can be simulated in our original definition by using the ι function (Def. 2, Section 2), which defines a particular initial state for each site.

4.2 An Implementation of `MPC`

For illustration purposes, we give a basic implementation of `MPC` in Figure 2. The idea is to let Site 1 decide on the order of all update operations. In general, the consensus mechanism can be arbitrary, and symmetric with respect to sites, but we present this one for its simplicity.

Though this is not the case in the model we presented in Section 2, we assume here that messages are received in the same order they were broadcast. More precisely, if one site broadcasts two messages, then every site will receive them in the order in which they were broadcast. In general, this can be implemented by adding a local version number to each broadcast message.

Upon receiving an update (L20), Site i with $i > 1$ forwards the update to Site 1. When receiving an update (L16) or when receiving a forwarded message (L25), Site 1 updates its local state, and broadcasts an `Apply` messages for the other sites. Finally, when receiving an `Apply` messages (L30), Site i with $i > 1$, updates its local state.

The query handler of each site (L35) simply returns the local state. This implementation ensures three properties that we formalize in the next section.

- **Monotonicity:** The list (maintained in the local state Q) of a site grows over time.
- **Prefix:** At any moment, given two lists l_1 and l_2 of two sites, l_1 is a prefix of l_2 or vice versa.
- **Consistency:** The list of a site only contains values that come from some update.

4.3 Formal Definition of `MPC`

Definition 9. *Given two lists $\ell_1, \ell_2 \in \mathbb{N}^*$, we say that ℓ_1 is a* prefix *of ℓ_2, denoted $\ell_1 \sqsubseteq \ell_2$, if there exists $\ell_3 \in \mathbb{N}^*$ such that $\ell_2 = \ell_1 \cdot \ell_3$. Moreover, ℓ_1 is a* strict prefix *of ℓ_2, denoted $\ell_1 \sqsubset \ell_2$, if $\ell_1 \sqsubseteq \ell_2$ and $\ell_2 \neq \ell_1$.*

By abuse of notation, we extend the prefix order to elements of Ans, which are of the form $\texttt{read}(\ell)$ where ℓ is a list (see Def. 1). Moreover, we also use the prefix notations for other types of sequences, such as executions. We now formally define `MPC`.

Definition 10. *`MPC` is the set of traces (t_r, W) where $t_r = (A, <, \mathsf{label})$ satisfying the following conditions:*

- ***Monotonicity:*** *A query aid' done after aid on the same site cannot return a smaller list. For all $aid, aid' \in A$, if $aid < aid'$, then $\mathsf{label}(aid) \sqsubseteq \mathsf{label}(aid')$.*
- ***Prefix:*** *Queries done on different sites are compatible, in the sense that one is a prefix of the other. For any all $aid, aid' \in A$, $\mathsf{label}(aid) \sqsubseteq \mathsf{label}(aid')$ or $\mathsf{label}(aid') \sqsubseteq \mathsf{label}(aid)$.*
- ***Consistency:*** *Queries only return elements that come from a write. For all $aid \in A$, and for any element $d \in Nat$ of $\mathsf{label}(aid)$, we have $d \in W$.*

The set of traces generated by the implementation in Figure 2 is exactly `MPC`.

4.4 Relation between MPC and Blockchains

In practice, the traces that the Bitcoin protocol [7] produces are traces which belong to MPC with high probability. This was shown in [8,9]. More precisely, they proved that the blockchains of two honest participants are compatible, in the sense that one should be a prefix of the other with high probability, when ignoring the last blocks[4]. This property is called *consistency* in [8], and it corresponds to the Prefix property we give in Section 4.3.

Moreover, it was shown [8,9] that the blockchain of an honest participant only grows over time. This property is called *future-self consistency* in [8], and it corresponds to the Monotonicity property we give in Section 4.3.

5 Nothing Stronger Than MPC in a Distributed Setting

We now proceed to our main result, stating that there exists no *convergent* implementation stronger than MPC. Convergent in our setting means that every write action performed should *eventually* be taken into account by all sites. We formalize this notion in Section 5.1.

We focus on convergent implementations in order to avoid trivial implementations that do not provide progress guarantees. For instance, implementations that do not communicate and always return the empty list for all queries are not convergent.

In Section 5.2, we prove several lemmas that hold for all implementations. We make use of these lemmas to prove our main theorem in Section 5.3.

5.1 Convergence Property

Convergence is formalized using the notion of eventual consistency (see e.g. [11,12] for definitions similar to the one we use there). A trace is eventually consistent if every write is *eventually* propagated to all sites. More precisely, for every action $\texttt{write}(d)$, the number of queries that do not contain d in their list must be finite. Note that this implies that all finite traces are eventually consistent.

Definition 11. *A trace (t_r, W) with $t_r = (A, <, \mathsf{label})$ is* eventually consistent *if for every $d \in W$, the set $\{aid \in A \mid d \notin \mathsf{label}(aid)\}$ is finite. An implementation is* convergent *if all of its traces are eventually consistent.*

5.2 Implementations Properties

We give a few lemmas that describe closure properties of the set of executions generated by implementations in our setting. The following lemma states that the implementation is *available for updates*, meaning that, given a finite execution, it is always possible to perform a new write at the end of the execution.

[4] In Bitcoin-like protocols, the most recent blocks are ignored as they are considered unsafe to use until newer blocks are appended after them.

Lemma 1 (Update Availability). *Let $\mathcal{I}$ be an implementation. Let e be a finite execution in $[\![\mathcal{I}]\!]$, and let $(t_r, W) = \mathsf{tr}(e)$. Let $d \in Nat$. Then, there exists an execution $e' \in [\![\mathcal{I}]\!]$ such that e is a prefix of e' and $\mathsf{tr}(e') = (t_r, W \cup \{d\})$.*

Proof. Since $e \in [\![\mathcal{I}]\!]$, we know that $e|_{pid}$ follows $\mathcal{I}$ and that there exists a run $q_0, \ldots, q_n$ for $e|_{pid}$. Let $(q_{n+1}, msg) = \mathsf{update\_handler}(q_n, \mathtt{write}(d))$. We distinguish two cases:

(1) If $msg = \perp$, let $e' = e \cdot (aid, pid, \mathtt{write}(d))$, where $aid \in \mathbb{N}$ is a fresh action identifier that does not appear in e, and pid is any process identifier in $\mathbb{P}$.

(2) If $msg \in \mathsf{Msg}$, let $e' = e \cdot (aid_1, pid, \mathtt{write}(d)) \cdot (aid_2, \mathtt{broadcast}(mid, msg))$, where aid_1 and aid_2 are fresh action identifiers from $\mathbb{N}$, and mid is a fresh message identifier.

In both cases, we prove that e' belongs to $[\![\mathcal{I}]\!]$ by adding q_{n+1} at the end of the run (once in case 1 or twice in case 2). Moreover, we have $\mathsf{tr}(e') = (t_r, W \cup \{d\})$, which concludes our proof. □

The next lemma shows that the implementation is *available for queries.* This means that given a finite execution, we can perform a query on any site and obtain an answer.

Lemma 2 (Query Availability). *Let $\mathcal{I}$ be an implementation. Let $e \in [\![\mathcal{I}]\!]$ be a finite execution and $pid \in \mathbb{P}$. Then, there exist $aid \in \mathbb{N}$ and $\ell \in \mathbb{N}^*$ such that the execution $e' = e \cdot (aid, pid, \mathtt{read}(\ell))$ belongs to $[\![\mathcal{I}]\!]$.*

Proof. Similar to the proof of Lemma 1, but using the query handler, instead of the update handler. This proof is also simpler, as there is no need to consider messages, since the query handler cannot broadcast any message. Therefore, in this proof, only case 1 needs to be considered. □

Then, we prove that it is possible to remove a finite number of query actions from any finite or infinite execution.

Lemma 3 (Invisible Reads). *Let $\mathcal{I}$ be an implementation. Let $e \in [\![\mathcal{I}]\!]$ be an execution (finite of infinite) of the form $e_1 \cdot (aid, pid, \mathtt{read}(\ell)) \cdot e_2$, where $aid \in \mathbb{N}$, $pid \in \mathbb{P}$ and $\ell \in \mathbb{N}^*$. Then, $e_1 \cdot e_2 \in [\![\mathcal{I}]\!]$.*

Proof. This is a direct consequence of Definition 5, which specifies that query actions do not modify the local state of sites, and do not broadcast messages. □

Finally, we prove a property about convergent implementations. We prove in Lemma 5 that given any finite execution e, it is always possible to add a query action that returns a list containing all the elements appearing in a write action of e. The proof relies on the notion of *limit* (as an infinite execution) of an infinite sequence of finite executions, and on Lemma 4, which shows that, under a fairness condition, the limit of executions in $[\![\mathcal{I}]\!]$ also belongs to $[\![\mathcal{I}]\!]$.

Definition 12. *Given an infinite sequence of finite sequences $e_1 \ldots, e_n, \ldots,$ such that for all $i \geq 1$, $e_i \sqsubset e_{i+1}$, the* limit *e^∞ of $e_1 \ldots, e_n, \ldots$ is the (unique) infinite sequence such that for all i, $e_i \sqsubset e^\infty$.*

Lemma 4 (Limit). *Let $\mathcal{I}$ be an implementation. Let $e_1 \ldots, e_n, \ldots$ be an infinite sequence of finite executions, such that for all $i \geq 1$, $e_i \in [\![\mathcal{I}]\!]$, $e_i \sqsubset e_{i+1}$, and such that for all $i \geq 1$, for all broadcast actions in e_i, and for all $pid \in \mathbb{P}$, there exists $j \geq 1$ such that e_j contains a corresponding receive action.*

Then, the limit e^∞ of $e_1 \ldots, e_n, \ldots$ belongs to $[\![\mathcal{I}]\!]$.

Proof. According to Definition 6, we have three points to prove. (1) (Projection) First, we want to show that, for all $pid \in \mathbb{P}$, the projection $e^\infty|_{pid}$ follows $\mathcal{I}$. For all $i \geq 1$, we know that $e_i \in [\![\mathcal{I}]\!]$, and deduce that $e_i|_{pid}$ follows $\mathcal{I}$. Let r_i be the run of $e_i|_{pid}$. Note that for all $i \geq 1$, we have $r_i \sqsubset r_{i+1}$. Let r^∞_{pid} be the limit of the runs $r_1, \ldots, r_n, \ldots$ By construction, r^∞_{pid} is a run of $e^\infty|_{pid}$, which shows that $e^\infty|_{pid}$ follows $\mathcal{I}$.

(2) (Causality) We need to prove that every receive action σ in e^∞ has a corresponding broadcast action σ' that precedes it in e^∞. Let e_i be a prefix of e^∞ that contains σ. Since $e_i \in [\![\mathcal{I}]\!]$, we know that there exists a broadcast action σ' corresponding to σ, and that precedes σ in e_i. Finally, since $e_i \sqsubset e^\infty$, σ' precedes σ in e^∞.

(3) (Fairness) We want to prove that for every broadcast action σ of e^∞ and for every site $pid \in \mathbb{P}$, there exists a corresponding receive action σ'. Let e_i be a prefix of e^∞ that contains σ. By assumption of the current lemma, there exists $j \geq 1$ such that e_j contains a receive action σ' corresponding to σ. Moreover, since $e_j \sqsubset e^\infty$, σ' belongs to e^∞, which concludes our proof. □

Lemma 5 (Convergence). *Let $\mathcal{I}$ be a convergent implementation. Let $e \in [\![\mathcal{I}]\!]$ be a finite execution and $pid \in \mathbb{P}$. Let $W \subseteq \mathbb{N}$ be the set of elements appearing in an update action of e, i.e., $W = \{d \in \mathbb{N} \mid \exists(aid, pid, \texttt{write}(d)) \in e\}$.*

Then, e can be extended in an execution $e \cdot e' \cdot (aid, pid, \texttt{read}(\ell)) \in [\![\mathcal{I}]\!]$ where $\ell \in \mathbb{N}^*$ contains every element of W, i.e., $W \subseteq \{d \in \mathbb{N} \mid d \in \ell\}$. Moreover, we can define such an extension e' that does not contain any query or update actions.

Proof. We build an infinite sequence of finite executions $e_1, \ldots, e_n, \ldots$, where for every $i \geq 1$, $e_i \in [\![\mathcal{I}]\!]$. Moreover, we have $e_1 = e$ and for every $i \geq 1$, $e_i \sqsubseteq e_{i+1}$, and e_{i+1} is obtained from e_i as follows.

For every broadcast action $(aid_1, pid_1, \texttt{broadcast}(mid, msg))$ in e_i, and for every $pid_2 \in \mathbb{P}$, if there is no receive action $(\_, pid_2, \texttt{receive}(mid, msg))$ in e_i, then we add one when constructing e_{i+1}. Moreover, if the message handler specifies that a message msg' should be sent when msg is received, we add a new broadcast action that sends msg', immediately following the receive action. Finally, using Lemma 2, we add a query action (`read`) on site pid.

Then, we define e^∞ to be the limit of $e_1, \ldots, e_n, \ldots$ By Lemma 4, we have $e^\infty \in [\![\mathcal{I}]\!]$. Since $\mathcal{I}$ is convergent, we know that e^∞ is eventually consistent. This ensures that for every $d \in W$, out of the infinite number of queries that belong to e^∞, only finitely many do not contain d.

Therefore, there exists $i \geq 1$ such that e_i ends with a query action that contains every element of W. By construction, e_i is of the form $e \cdot e'' \cdot (aid, pid, \texttt{read}(\ell))$. Using Lemma 3, we remove every query action that appears in e'', and obtain an execution of the form $e \cdot e' \cdot (aid, pid, \texttt{read}(\ell))$ where $\ell \in \mathbb{N}^*$ contains every element of W, and where e' does not contain any query or update actions. □

5.3 Nothing Is Stronger Than `MPC` in a Distributed Setting

We now proceed with the proof that no convergent implementation is strictly stronger than `MPC`. We start with an implementation $\mathcal{I}$ that is strictly stronger than `MPC` and derive a contradiction.

More precisely, using the lemmas proved in Section 5.2, we prove that any trace of `MPC` belongs to $\mathsf{tr}(\llbracket\mathcal{I}\rrbracket)$. First, we show that this holds for finite traces, by using an induction on the number of write operations in the trace. Then, we extend the proof to infinite traces by going to the limit.

Theorem 1. *Let $\mathcal{I}$ be a convergent implementation. Then, $\mathcal{I}$ is not strictly stronger than `MPC`: $\mathcal{I} \not\prec$ `MPC`.*

Proof. Assume that $\mathcal{I}$ is strictly stronger than `MPC` i.e. $\mathcal{I} \prec$ `MPC`. Our goal is to prove that `MPC` $\preceq \mathcal{I}$ therefore leading to a contradiction. In terms of traces, we want to prove that `MPC` $\subseteq \mathsf{tr}(\llbracket\mathcal{I}\rrbracket)$.

Let $t = (t_r, W) \in$ `MPC`. Our goal is to prove that $t \in \mathsf{tr}(\llbracket\mathcal{I}\rrbracket)$.

Case where t is finite. We prove this part by induction in Lemma 6.

Case where t is infinite. Let $t_r = (A, <, \mathsf{label})$. We first order all the query actions in A as a sequence $aid_1, \dots, aid_n, \dots$ such that for every $i \geq 1$, $\mathsf{label}(aid_i) \sqsubseteq \mathsf{label}(aid_{i+1})$, and for every $i, j \geq 1$, $aid_i < aid_j$ (in the program order of t_r) implies $i < j$. Defining such a sequence is possible thanks to the Monotonicity property of `MPC`.

For each $i \geq 1$, we define a *finite* trace t_i that contains all query actions aid_j with $j \leq i$, and the subset W_i of W that contains all elements appearing in these query actions, i.e. $W_i = \{d \in W \mid d \in \mathsf{label}(aid_i)\}$. Our goal is to construct an execution $e_i \in \llbracket\mathcal{I}\rrbracket$ such that $\mathsf{tr}(e_i) = t_i$, and such that for all $i \geq 1$, $e_i \sqsubset e_{i+1}$. We then define e^∞ as the limit of $e_1, \dots, e_n, \dots$ By Lemma 4, we have $e^\infty \in \llbracket\mathcal{I}\rrbracket$. Since $\mathsf{tr}(e^\infty) = t$, we deduce that $t \in \mathsf{tr}(\llbracket\mathcal{I}\rrbracket)$, which concludes the proof.

We now explain how to construct e_i, for every $i \geq 1$, by induction on i. Let e_0 be the empty execution and $t_0 = \mathsf{tr}(e_0)$. For $i \geq 0$, we define e_{i+1} by starting from e_i, and extending it as follows. By induction, we know that $\mathsf{tr}(e_i) = t_i$, and want to extend it into an execution e_{i+1} such that $\mathsf{tr}(e_{i+1}) = t_{i+1}$.

(Similar to Lemma 5) For every broadcast action $(aid_1, pid_1, \mathtt{broadcast}(mid, msg))$ in e_i, and for every $pid_2 \in \mathbb{P}$, if there is no receive action $(\_, pid_2, \mathtt{receive}(mid, msg))$ in e_i, then we add one when constructing e_{i+1}. Moreover, if the message handler specifies that a message msg' should be sent when msg is received, we add a new broadcast action that sends msg', immediately following the receive action.

Then, similarly to the construction in Lemma 6, we add update and query actions (using lemmas 1, 2, and 5) in order to obtain an execution e_{i+1} such that $\mathsf{tr}(e_{i+1}) = t_{i+1}$. □

Lemma 6 below, used in Theorem 1, shows that no convergent implementation can produce strictly fewer finite traces than `MPC`.

Lemma 6. *Let $\mathcal{I}$ be a convergent implementation such that $\mathcal{I} \prec$ `MPC`, and let t be a finite trace of `MPC`. Then, there is a finite execution $e \in \llbracket\mathcal{I}\rrbracket$ such that $\mathsf{tr}(e) = t$.*

Proof. Let $t = (t_r, W)$. We proceed by induction on the size of W, denoted n.

Case $n = 0$. In that case, the set W is empty. First, by definition of $[\![\mathcal{I}]\!]$, we have $\varepsilon \in [\![\mathcal{I}]\!]$ where ε is the empty execution. Then, for each read operation in t, and using Lemma 2, we add a read operation to the execution. We obtain an execution $e \in [\![\mathcal{I}]\!]$.

We then have to prove that $\mathtt{tr}(e) = t$, meaning that all the read operations of e return the empty list, as in t. By our assumption that $\mathcal{I} \prec \mathtt{MPC}$, we know that $\mathtt{tr}(e) \in \mathtt{MPC}$. By definition of MPC, and since e contains no write operation, the Consistency property of MPC ensures that all the read actions of e return the empty list. Therefore, we have $\mathtt{tr}(e) = t$, which concludes our proof.

Case $n > 0$. We consider two subcases. (1) There exists a write $w \in W$ whose value does not appear in t_r. We consider the trace $t' = (t_r, W \setminus \{w\})$. By definition of MPC, t' belongs to MPC, and we deduce by induction hypothesis that there exists an execution $e' \in [\![\mathcal{I}]\!]$ such that $\mathtt{tr}(e') = t'$. By Lemma 1, we extend e' in an execution $e \in [\![\mathcal{I}]\!]$ so that $\mathtt{tr}(e) = t$, which is what we wanted to prove.

(2) All the writes of W appear in the reads of t_r. By the Consistency and Prefix properties of MPC, there exists a non-empty sequence $\ell \in \mathbb{N}^+$ of elements from W, such that all read actions return a prefix of ℓ, and there exist read actions that return the whole list ℓ.

Let $\ell = \ell' \cdot d$, where $d \in \mathbb{N}$ is the last element of ℓ. Let t' be the trace $(t'_r, W \setminus \{d\})$, such that t'_r is the trace t_r where every query action labelled by ℓ is replaced by a query action labelled by ℓ', and implicitly, every query action labelled by any prefix of ℓ' is unchanged. Let R the set of the newly added query actions, and let $P \subseteq \mathbb{P}$ be the set of site identifiers that appear in an action of R.

By definition of MPC, we have $t' \in \mathtt{MPC}$. By induction hypothesis, we deduce that there exists a finite execution $e' \in [\![\mathcal{I}]\!]$ such that $\mathtt{tr}(e') = t'$.

Then, by Lemma 1, we add at the end of e' an update action (on some site $pid \in \mathbb{P}$ and with some fresh $aid \in \mathbb{N}$), which is of the form $(aid, pid, \mathtt{write}(d))$, so we get an execution $e'' \in [\![\mathcal{I}]\!]$ such that $\mathtt{tr}(e'') = (t'_r, W \setminus \{d\} \cup \{d\}) = (t'_r, W)$.

Using Lemma 5, we extend e'' in an execution e''' by adding queries to the sites in P, as many as were replaced by queries in R. Since $\mathcal{I} \prec \mathtt{MPC}$, and since by Lemma 5, the answers to these queries must contain all the elements of ℓ, we conclude that the only possible answer for all these queries is the entire list ℓ.

Finally, we use Lemma 3 to remove the queries R from e''', and we obtain an execution in $[\![\mathcal{I}]\!]$ whose trace is t. □

6 Comparison with Other Consistency Criteria

6.1 Relation between MPC and other consistency criteria

Consistency criteria are usually defined in terms of *full traces* that contain both the read and write operations in the program order (see e.g. [11]). The definition of trace we used in this paper (Def. 7, Section 3) puts the writes in an unordered set, unrelated to the read operations. This choice is justified in large-scale, open, implementations, such as the Bitcoin blockchain. Indeed, in these systems, any participant can perform a write operation (e.g., a Bitcoin transaction), and the origin of the write has no relevance for the protocol.

When considering full traces, `MPC` as a consistency criterion is strictly weaker than strong consistency. Indeed, `MPC` allows a trace where a read preceded by a write on the same site ignores that write.

As explained in the introduction, `MPC` is not comparable to causal consistency. `MPC` allows full traces that causal consistency forbids and vice versa. Therefore, our result stating that nothing stronger than `MPC` that can be implemented in a distributed setting does not contradict earlier results of [13] and [5], which show that nothing stronger than variants of causal consistency can be implemented.

6.2 Relation with other criteria when using our notion of trace

When using our notion of trace, `MPC` is strictly stronger than causal consistency. First, `MPC` is stronger than causal consistency because every trace of `MPC` can be produced by a causally consistent system. The main reason is that our notion of trace doesn't capture any causality relation. Moreover, there are some traces that causal consistency produces and which do not belong to `MPC`, e.g. a trace where Site 1 has a $\texttt{read}[1,2]$ operation, and Site 2 has a $\texttt{read}[2,1]$, where $\texttt{write}(1)$ and $\texttt{write}(2)$ are not causally related (this explains that `MPC` is *strictly* stronger than causal consistency).

Moreover, it is interesting to note that, for our notion of trace, the traces allowed by `MPC` are exactly the traces allowed by strong consistency. This entails that, if the replicated data structure is used by clients who can only observe our traces, then there is no need to implement strong consistency. In short, `MPC` and strong consistency are indistinguishable to these clients.

7 Conclusion

We have investigated the question of what is the strongest consistency criterion that can be implemented when replicating a data structure, in distributed systems under availability and partition-tolerance requirements. Earlier work had established the impossibility of implementing strong consistency in such a system model, but left open the question of the strongest criteria that *can* be implemented. In this paper we have focused on the *Monotonic Prefix Consistency* (`MPC`) criterion. We proposed an implementation of `MPC` and showed that no criterion stronger than `MPC` can be implemented. Importantly, Blockchain protocols, such as Bitcoin, implement `MPC` with high probability, and therefore come as close as possible to strong consistency.

In future work we plan to investigate how the strongest achievable consistency criterion depends on observability – that is, the information encoded in a trace – and study conditions for the (non)existence of a strongest consistency criterion. We are also interested in extending our result to other system models. Specifically, answering the question of what is the strongest consistency criterion that can be implemented in systems where updates are not anonymous, or where the system is permissioned, i.e., where different sites may have different roles, such as primary-backup replication schemes.

References

1. Herlihy, M., Wing, J.M.: Linearizability: A correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. **12**(3) (1990)
2. Brewer, E.: Cap twelve years later: How the "rules" have changed. Computer **45**(2) (2012)
3. Gilbert, S., Lynch, N.A.: Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. SIGACT News **33**(2) (2002) 51–59
4. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM **21**(7) (July 1978) 558–565
5. Attiya, H., Ellen, F., Morrison, A.: Limitations of highly-available eventually-consistent data stores. IEEE Transactions on Parallel and Distributed Systems **28**(1) (2017) 141–155
6. Terry, D.: Replicated data consistency explained through baseball. Technical Report MSR-TR-2011-137, Microsoft Research (October 2011)
7. Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system (2008)
8. Pass, R., Seeman, L., Shelat, A.: Analysis of the blockchain protocol in asynchronous networks. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques, EUROCRYPT'17. Volume 10211 of Lecture Notes in Computer Science., Paris, France (April 2017) 643–673
9. Garay, J., Kiayias, A., Leonardos, N.: The bitcoin backbone protocol: Analysis and applications. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques, Springer (2015) 281–310
10. Guerraoui, R., Pavlovic, M., Seredinschi, D.A.: Trade-offs in replicated systems. IEEE Data Engineering Bulletin **39** (2016) 14–26
11. Burckhardt, S.: Principles of Eventual Consistency. Now Publishers (October 2014)
12. Bouajjani, A., Enea, C., Hamza, J.: Verifying eventual consistency of optimistic replication systems. In Jagannathan, S., Sewell, P., eds.: The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'14, San Diego, CA, USA, January 20-21, 2014, ACM (2014) 285–296
13. Mahajan, P., Alvisi, L., Dahlin, M.: Consistency, availability, convergence. Technical Report TR-11-22, Computer Science Department, University of Texas at Austin (May 2011)