Monotonic Prefix Consistency in Distributed Systems

Alain Girault, Gregor Gössler, Rachid Guerraoui, Jad Hamza, Dragos-Adrian Seredinschi

Monotonic Prefix Consistency 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 distributed systems. Specifically, we consider a distributed system that replicates its data at multiple sites, which is prone to partitions, and which is assumed to be available (in the sense that queries are always eventually answered). 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 available distributed systems.

We focus on determining the strongest consistency criterion that can be implemented in a convergent and available distributed system that tolerates partitions. We focus on objects where the set of operations can be split into updates and queries. We show that no criterion stronger than Monotonic Prefix Consistency (MPC) can be implemented.

1 Introduction

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

An ideal replication scheme could be implemented by keeping all sites synchronized after each update to the data type. 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 type 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* [2,3,4]. High availability (hereafter *availability* for short) means that sites must answer users' requests directly, without waiting for outside communication.

Given this impossibility, developers rely on weaker notions of consistency, such as causal consistency [5]. 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 type 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 type. 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 type? We focus in this paper on data types where the set of operations can be split into two disjoint sets, updates and queries. Updates modify the state and but do not return values, while queries return values without modifying the state.

In [4], 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, [4] 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 type.

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 (availability and convergence), nothing stronger than *monotonic prefix consistency* (`MPC`) [6] can be implemented. This result does not contradict the result from [4], 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.

Overall, our contribution is to prove that, for a notion of behaviors where the time and place of origin of updates do not matter, nothing stronger than `MPC` can be implemented in a distributed setting. Moreover, we remark that clients that only have the observability defined in Section 3 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 are 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 observability mentioned above, and under conditions natural in a large-scale network (availability, convergence), nothing stronger than `MPC` can be implemented (Section 6). Then we compare `MPC` with other consistency models (Section 7), and conclude (Section 8).

To improve the presentation, some proofs are deferred to the appendix.

2 Replicated Implementations

An *implementation* of a replicated data type 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 type, 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 (hypothetical) clients to submit updates to the data type. 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 type. 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 type*. The list supports an update operation of the form $\texttt{write}(d)$, with $d \in \mathbb{N}$, which adds the element d to the end of 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 type because queries return the history of all updates that ever happened. In that regard, lists can encode any other data type 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 types 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 total 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 total 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 total 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 an *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* $(aid, pid, \mathtt{broadcast}(mid, msg))$*, and a* receive action *is a tuple* $(aid, pid, \mathtt{receive}(mid, msg))$*, 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* $(aid, pid, \mathtt{write}(d))$ *where* $d \in \mathbb{N}$*. Finally, a* query action *or a* read action *is a tuple* $(aid, pid, \mathtt{read}(\ell))$ *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* $\sigma_1 \ldots \sigma_n \ldots$ *from site pid* follows $\mathcal{I}$ *if there exists a sequence of states* $q_0 \ldots q_n \ldots$ *such that* $q_0 = \iota(pid)$*, and for all* $i \in \mathbb{N}\backslash\{0\}$*, we have:*

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

Remark 2. The implementations we consider are *available* by construction, in the sense that any site allows any updates or queries to be done at any time, and answers to queries directly. This is ensured by the fact that our update and query handlers are total functions. More precisely, the item 1 of Definition 5 (together with Definition 6) ensures that updates can be performed at any time through the update handler (*update availability*).

The broadcast action that happens right after an update action must be thought of as happening right after the update. Broadcast actions do not involve actively waiting for responses, and as such do not prevent availability.

Similarly, the item 2 of Definition 5 ensures that any query of any site is answered immediately, only using the local state of the site (*query availability*). We later formalize this in Lemmas 1 and 2.

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, as 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 a *trace*. Informally, the trace of an execution corresponds to what is observable from the point of view of clients using the data type.

Our notion of a 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) The origin of updates is not relevant from a client's perspective. This models publicly accessible data structures where any client 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 updates (without their site identifiers), and records for each site the sequence of queries that happened on this site.

Definition 7. *A* trace (t_r, W) *is a pair where* t_r *is a labelled partially ordered set (see hereafter for more details), and* W *is a subset of* $\mathbb{N}$. *The trace* (t_r, W) *corresponding to an execution* e *is denoted* $\mathsf{tr}(e)$, *where* $t_r = (A, <, \mathsf{label})$ *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;*
- *$\mathsf{label} : A \to \mathsf{Ans}$ is the* labelling function *such that for any $aid \in A$, $\mathsf{label}(aid)$ 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.

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*

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

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

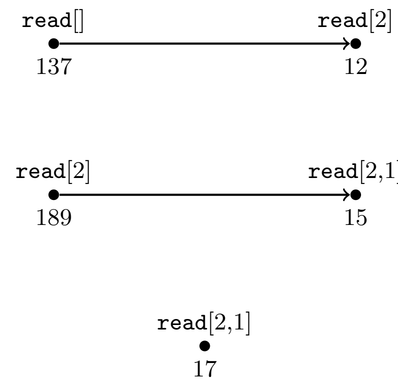

Fig. 1: An execution e read from top to bottom, then left to right $(188, 3713, \ldots, 189, 733, \ldots, 15)$ and its corresponding trace $\mathsf{tr}(e) = (t_r, W)$ (right). The bullets represent the action identifiers of t_r (written under the bullet), and the corresponding labels are represented right above. The arrows represent the program order $<$ of t_r. The set of writes is W is {1,2,3} (from actions 152, 37, and 188 respectively).

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 find an implementation $\mathcal{I}$ which is minimal in the $\preceq$ ordering, i.e., for which there does not exist an implementation $\mathcal{I}'$ strictly stronger than $\mathcal{I}$.

4 Definition of Monotonic Prefix Consistency (MPC)

Often called consistent prefix [6,7], the MPC model requires that all sites of the replicated system agree on the order of write operations (i.e., updates on the state). More precisely, this means that given two read operations (possibly on two different sites), one read has to return a list of writes which is a prefix of the other. 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. The trace given in Figure 1 satisfies these constraints.

Note that the 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 order in which these updates apply at all sites. Moreover, `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.

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 $\ell_1 \neq \ell_2$.*

By abuse of notation, we extend the prefix order to elements of Ans, which are of the form $\mathtt{read}(\ell)$ 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 $d \in \mathbb{N}$ of $\mathsf{label}(aid)$, we have $d \in W$.*

5 Feasibility of MPC

In this section, we provide a toy implementation (Figure 2) whose traces are all in `MPC`, to show that `MPC` is indeed implementable. The idea is to let Site 1 decide on the order of all update operations. In general, the consensus mechanism for implementing `MPC` can be arbitrary, and symmetric with respect to sites, but we present this one for its simplicity.

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

Each site maintains a local state (in Q) which is the prefix of updates as decided by Site 1. Upon receiving an update (line 16), Site i with $i > 1$ forwards the update to Site 1. When receiving an update (line 12) or when receiving a forwarded message (line 20), Site 1 updates its local state, and broadcasts an `Apply` messages for the other sites. Finally, when receiving an `Apply` messages (line 25), Site i with $i > 1$, updates its local state.

We assume that the `Apply` messages sent by Site 1 are received in the same order in which they are sent, which can be implemented by having Site 1 add a local version number to each broadcast message, and having sites with $i > 1$

```
1 // Each site stores an element of Q, defined as a list of numbers
2 type Q = List[Nat]
3
4 abstract class Msg
5 // Forwarded messages go from Site i to Site 1, for all i > 1
6 case class Forwarded(d: Nat) extends Msg
7 // Apply messages originate from Site 1 and go to Site i, for i > 1
8 case class Apply(d: Nat) extends Msg
9
10 // The update handler for Site 1 appends element 'upd' to q,
11 // and tells the other sites to do the same with Apply(upd)
12 def update_handler(q: Q, upd: Upd) = (append(q,upd), Apply(upd))
13
14 // The update handler for Site i > 1 sends a message Forwarded(upd)
15 // which is destined for Site 1, and does not modify the state
16 def update_handler(q: Q, upd: Upd) = (q, Forwarded(upd))
17
18 // Message handler for Site 1 (ignores Apply messages)
19 def msg_handler(msg: Msg) = msg match {
20   case Forwarded(d) => (append(q,d), Apply(upd))
21 }
22
23 // Message handler for Site i > 1 (ignores Forwarded messages)
24 def msg_handler(msg: Msg) = msg match {
25   case Apply(d) => (append(q,d), ⊥)
26 }
27
28 // The query handler of any site returns the local state
29 def query_handler(q: Q) = q
```

Fig. 2: An implementation of MPC which is centralized at Site 1.

cache messages until all previous messages have been received. Similarly, we assume that each message which is sent by a site is treat at most once by each of the other sites. We omit these details in Figure 2. Finally, the query handler of each site (line 29) simply returns the list maintained in the local state.

We now prove that all the traces of the implementation described in Figure 2 satisfy MPC.

Proposition 1. *Let $\mathcal{I}$ be the implementation of Figure 2. Then $\mathcal{I} \preceq$ MPC.*

The formal proof is in Appendix A. It relies on the observation that the implementation maintains the following invariant:

- (Related to **Monotonicity**) The list maintained in the local state Q of each site grows over time.
- (Related to **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. Any list is always a prefix of (or equal to) the list of Site 1.
- (Related to **Consistency**) The list of a site only contains values that come from some update.

6 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 6.1. This convergence assumption prevents trivial implementations, for instances ones that do not communicate and always return the empty list for all queries.

In Section 6.2, we prove several lemmas that hold for all implementations. We make use of these lemmas to prove our main theorem in Section 6.3.

6.1 Convergence Property

Convergence is formalized using the notion of eventual consistency (see e.g. [8,9] 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.*

6.2 Properties of Implementations

Lemmas 1, 2, and 3 describe basic closure properties of the set of executions generated by implementations in our setting. The semantics described in Section 2 ensures that new updates and queries can always be performed following an existing execution. Moreover, queries never modify the state, and therefore removing a read action from an execution does not affect its validity (Lemma 3).

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 $d \in \mathbb{N}$. 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 by Definitions 5 and 6 that $e|_{pid}$ follows $\mathcal{I}$ and that there exists a run $q_0, \ldots, q_n$ for $e|_{pid}$. Let $(q_{n+1}, msg) = \mathsf{update\_handler}(q_n, \texttt{write}(d))$. We distinguish two cases:

(1) If $msg = \bot$, let $e' = e \cdot (aid, pid, \texttt{write}(d))$, 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 $e' = e \cdot (aid_1, pid, \texttt{write}(d)) \cdot (aid_2, \texttt{broadcast}(mid, msg))$, where aid_1, aid_2 are fresh action identifiers, and mid is a fresh message identifier.

In both cases, we construct a new run by adding the state q_{n+1} at the end of the run $q_0, \ldots, q_n$ (once in case 1, and twice in case 2). By Definition 5, this ensures that $e'|_{pid}$ follows $\mathcal{I}$, and we then obtain $e' \in [\![\mathcal{I}]\!]$ by Definition 6. Moreover, we have $\mathsf{tr}(e') = (t_r, W \cup \{d\})$, which concludes our proof. $\square$

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, as ensured by the definitions given in Section 2. The proof is in Appendix B.

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 $[\![\mathcal{I}]\!]$.*

We then prove it is possible to remove any query action from an execution.

Lemma 3 (Invisible Reads). *Let $\mathcal{I}$ be an implementation. Let $e \in [\![\mathcal{I}]\!]$ be an execution (finite or 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, $e_1 \cdot e_2 \in [\![\mathcal{I}]\!]$.*

Lemma 4 shows that, given an infinite sequence of increasing finite executions $e_1 \ldots, e_n, \ldots$ that satisfy a fairness condition, the *limit* execution (which is infinite) also belongs to $[\![\mathcal{I}]\!]$. The fairness condition states that each broadcast that appears in an execution e_i must have corresponding receive actions for each of the other sites $pid \in \mathbb{P}$ in some executions e_j.

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 $[\![\mathcal{I}]\!]$.

We finally prove in Lemma 5 that, given any finite execution e, it is possible to add a query action that returns a list containing all the elements W appearing in some write action of e. The proof relies on extending e into an infinite execution e^∞ with an infinite number of queries. Our convergence assumption then ensures that only finitely many of those queries can ignore W (that is, return a list that does not contain all elements of W). This shows that there exists a query operation (actually, infinite many) in e^∞ that returns a list containing all elements of W. We can therefore take the finite prefix of e^∞ that ends with this query operation.

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., $W = \{d \in \mathbb{N} \mid \exists (aid, pid, \mathtt{write}(d)) \in e\}$.*

Then, e can be extended in an execution $e \cdot e' \cdot (aid, pid, \mathtt{read}(\ell)) \in [\![\mathcal{I}]\!]$ 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 ($\texttt{read}$) 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. $\square$

6.3 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 6.2, we prove that any trace of MPC belongs to $\mathsf{tr}([\![\mathcal{I}]\!])$. First, we show in Lemma 6 that this holds for finite traces, by using an induction on the number of write operations in the trace. For each write operation w, we apply Lemma 5 in order to force the sites to take into account w.

Lemma 6. *Let $\mathcal{I}$ be a convergent implementation such that $\mathcal{I} \prec \texttt{MPC}$, and let t be a finite trace of MPC. Then, there is a finite execution $e \in [\![\mathcal{I}]\!]$ 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 $\mathsf{tr}(e) = t$, meaning that all the read operations of e return the empty list, as in t. By our assumption that $\mathcal{I} \prec \texttt{MPC}$, we know that $\mathsf{tr}(e) \in \texttt{MPC}$. 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 $\mathsf{tr}(e) = t$, 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 $\texttt{MPC}$, t' belongs to $\texttt{MPC}$, and we deduce by induction hypothesis that there exists an execution $e' \in [\![\mathcal{I}]\!]$ such that $\mathsf{tr}(e') = t'$. By Lemma 1, we extend e' in an execution $e \in [\![\mathcal{I}]\!]$ so that $\mathsf{tr}(e) = t$, 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 $\texttt{MPC}$, 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 $\texttt{MPC}$, we have $t' \in \texttt{MPC}$. By induction hypothesis, we deduce that there exists a finite execution $e' \in [\![\mathcal{I}]\!]$ such that $\mathsf{tr}(e') = t'$.

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 $(aid, pid, \texttt{write}(d))$, so we get an execution $e'' \in [\![\mathcal{I}]\!]$ such that $\mathsf{tr}(e'') = (t'_r, W \setminus \{d\} \cup \{d\}) = (t'_r, W)$.

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 $\mathcal{I} \prec \texttt{MPC}$, 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. $\square$

We then extend Lemma 6 to infinite executions.

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

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

Let $t = (t_r, W) \in \texttt{MPC}$. We need to show that $t \in \mathsf{tr}([\![\mathcal{I}]\!])$.

Case where t is finite. Proven 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 $aid_1, \ldots, aid_n, \ldots$ 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 $\texttt{MPC}$.

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 $e_i \in [\![\mathcal{I}]\!]$ 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 $e_1, \ldots, e_n, \ldots$ By Lemma 4, we have $e^\infty \in [\![\mathcal{I}]\!]$. Since $\mathsf{tr}(e^\infty) = t$, we deduce that $t \in \mathsf{tr}([\![\mathcal{I}]\!])$, 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 $\mathtt{tr}(e_i) = t_i$, and want to extend it into an execution e_{i+1} such that $\mathtt{tr}(e_{i+1}) = t_{i+1}$.

The next step of the proof is similar to the proof of 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 $\mathtt{tr}(e_{i+1}) = t_{i+1}$. □

7 Comparison with Other Consistency Criteria

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., [8]). 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 blockchain protocols. Indeed, in these systems, any participant can perform a write operation (e.g., a blockchain 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 [10] and [4], which show that nothing stronger than variants of causal consistency can be implemented.

Relation with other criteria when using our notion of a trace. When using our notion of a 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 a trace does not capture any causality relation. Moreover, there are some traces that causal consistency produces and that do not belong to `MPC`, e.g. a trace where Site 1 has a $\mathtt{read}[1, 2]$ operation, Site 2 has a $\mathtt{read}[2, 1]$, and where $\mathtt{write}(1)$ and $\mathtt{write}(2)$ are not causally related as they happen at the *same time* (this explains that `MPC` is *strictly* stronger than causal consistency).

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

8 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.

It is worth noting that blockchain protocols, such as the Bitcoin protocol [11], implement `MPC` with high probability: the traces that the protocol produces are traces that belong to `MPC` with high probability. This was shown in [12,13]. More precisely, the authors 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 [12], and it corresponds to the Prefix property we give in Section 4. Moreover, it was shown [12,13] that the blockchain of an honest participant only grows over time. This property is called *future-self consistency* in [12], and it corresponds to the Monotonicity property we give in Section 4.

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 the origin of updates do matter for the protocol. Also, the question whether MPC is the strongest implementable consistency criterium in a *probabilistic* setting, remains open.

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. 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
5. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM **21**(7) (July 1978) 558–565
6. Terry, D.: Replicated data consistency explained through baseball. Technical Report MSR-TR-2011-137, Microsoft Research (October 2011)

[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.

7. Guerraoui, R., Pavlovic, M., Seredinschi, D.A.: Trade-offs in replicated systems. IEEE Data Engineering Bulletin **39** (2016) 14–26
8. Burckhardt, S.: Principles of Eventual Consistency. Now Publishers (October 2014)
9. 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
10. Mahajan, P., Alvisi, L., Dahlin, M.: Consistency, availability, convergence. Technical Report TR-11-22, Computer Science Department, University of Texas at Austin (May 2011)
11. Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system (2008)
12. 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
13. 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

A Proof of Feasability of MPC

Proposition 1. *Let $\mathcal{I}$ be the implementation of Figure 2. Then $\mathcal{I} \preceq$ MPC.*

Proof. Let $e \in [\![\mathcal{I}]\!]$, we establish an inductive invariant that holds for every finite prefix e' of e. Let A be the set of action identifiers of e'. Let $\ell \in \mathbb{N}^*$ be the sequence of values that appear in a broadcast message `Apply` from Site 1, in the order they appear in e'.

Let $\mathbb{P}$ be the set of process identifiers. For each site $pid \in \mathbb{P}$, consider the unique run r for the projection $e'|_{pid}$, and let $\ell_{pid} \in \mathbb{N}^*$ be the sequence maintained in the local state of Site pid at the end of the run r.

Let $t' = \mathsf{tr}(e')$ be the trace of e', with $t' = (t_r, W)$.

Then, we have the following properties.

1. For every $\texttt{Apply}(d)$ message with $d \in \mathbb{N}$ that appears in e' (from Site 1), we have $d \in W$.
2. For every $\texttt{Forwarded}(d)$ message with $d \in \mathbb{N}$ that appears in e' (from Site i with $i > 1$), we have $d \in W$.
3. The elements of ℓ are in W.
4. For every $pid \in \mathbb{P}$, $l_{pid} \sqsubseteq \ell$.
5. For every query $(\_, pid, \texttt{read}\ \ell')$ in e with $pid \in \mathbb{P}$, we have $\ell' \sqsubseteq \ell_{pid}$.
6. **Consistency:** For all $aid \in A$, and for any element $d \in \mathbb{N}$ of $\mathsf{label}(aid)$, we have $d \in W$.
7. **Prefix:** For any all $aid, aid' \in A$, $\mathsf{label}(aid) \sqsubseteq \mathsf{label}(aid')$ or $\mathsf{label}(aid') \sqsubseteq \mathsf{label}(aid)$.
8. **Monotonicity:** For all $aid, aid' \in A$, if $aid < aid'$, then $\mathsf{label}(aid) \sqsubseteq \mathsf{label}(aid')$.

We can see that this invariant holds for the empty execution, and that any action that the implementation can take maintains it.

B Closure Properties of Implementations

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 $e' = e \cdot (aid, pid, \texttt{read}(\ell))$ 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. □

Lemma 3 (Invisible Reads). *Let $\mathcal{I}$ be an implementation. Let $e \in [\![\mathcal{I}]\!]$ be an execution (finite or infinite) of the form $e_1 \cdot (aid, pid, \texttt{read}(\ell)) \cdot e_2$, 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. □

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. $\square$