D.1.2 – Modular quasi-causal data structures

Davide Frey, Roy Friedman, Achour Mostefaoui, Matthieu Perrin, Michel Raynal, François Taïani

D.1.2 – Modular quasi-causal data structures

ANR-13-INFR-0003

socioplug.univ-nantes.fr

Davide Frey[1,2], Roy Friedman[7], Achour Mostfaoui[3], Matthieu Perrin[3], Michel Raynal[1,2,5,6], Franois Taiani[1,2,4]

[1]*IRISA Rennes*
[2]*Inria Rennes - Bretagne Atlantique*
[3]*LINA, Universit de Nantes*
[4]*Universit de Rennes 1 - ESIR*
[5]*Universit de Rennes 1 - ISTIC*
[6]*Institut Universitaire de France*
[7]*The Technion, Israel*
Email contact: {`davide.frey,michel.raynal,francois.taiani`}`@irisa.fr`, `matthieu.perrin@etu.univ-nantes.fr`, `achour.mostefaoui@univ-nantes.fr`

In large scale systems such as the Internet, replicating data is an essential feature in order to provide availability and fault-tolerance. Attiya and Welch proved that using strong consistency criteria such as atomicity is costly as each operation may need an execution time linear with the latency of the communication network. Weaker consistency criteria like causal consistency and PRAM consistency do not ensure convergence. The different replicas are not guaranteed to converge towards a unique state. Eventual consistency guarantees that all replicas eventually converge when the participants stop updating. However, it fails to fully specify the semantics of the operations on shared objects and requires additional non-intuitive and error-prone distributed specification techniques. In addition existing consistency conditions are usually defined independently from the computing entities (nodes) that manipulate the replicated data; i.e., they do not take into account how computing entities might be linked to one another, or geographically distributed. In this deliverable, we address these issues with two novel contributions.

The first contribution proposes a notion of *proximity graph* between computing nodes. If two nodes are connected in this graph, their operations must satisfy a strong consistency condition, while the operations invoked by other nodes are allowed to satisfy a weaker condition. We use this graph to provide a generic approach to the hybridization of data consistency conditions into the same system. Based on this, we design a distributed algorithm based on this proximity graph, which combines sequential consistency and causal consistency (the resulting condition is called *fisheye consistency*).

The second contribution of this deliverable focuses on improving the limitations of eventual consistency. To this end, we formalize a new consistency criterion, called *update consistency*, that requires the state of a replicated object to be consistent with a linearization of all the updates. In other words, whereas atomicity imposes a linearization of all of the operations, this criterion imposes this only on updates. Consequently some read operations may return out-dated values. Update consistency is stronger than eventual consistency, so we can replace eventually consistent objects with update consistent ones in any program. Finally, we prove that update consistency is universal, in the sense that any object can be implemented under this criterion in a distributed system where any number of nodes may crash.

Keywords: Causal consistency, Sequential Consistency, Eventual Consistency, Scalability, Distributed Computer Systems

1 Introduction

Reliability of large scale systems is a big challenge when building massive distributed applications over the Internet. At this scale, data replication is essential to ensure availability and fault-tolerance. In a perfect world, distributed objects should behave as if there is a unique physical shared object that evolves following the atomic operations issued by the participants†. This is the aim of strong consistency criteria such as linearizability and sequential consistency. These criteria serialize all the operations so that they look as if they happened sequentially, but they are costly to implement in message-passing systems. If one considers a distributed implementation of a shared register, the worst-case response time must be proportional to the latency of the network either for the reads or for the writes to be sequentially consistent [LS88] and for all the operations for linearizability [AW94]. This generalizes to many objects [AW94]. Moreover, the availability of the shared object cannot be ensured in asynchronous systems where more than a minority of the processes of a system may crash [ABD95]. In large modern distributed systems such as Amazon's cloud, partitions do occur between data centers, as well as inside data centers [Vog08]. Moreover, it is economically unacceptable to sacrifice availability. The only solution is then to provide weaker consistency criteria. Several weak consistency criteria have been considered for modeling shared memory such as PRAM [LS88] or causality [ANB+95]. They expect the local histories observed by each process to be plausible, regardless of the other processes. However, these criteria do not impose that the data eventually converges to a consistent state. Eventual consistency [Vog08] is another weak consistency criterion which requires that when all the processes stop updating then all replicas eventually converge to the same state.

These weaker consistency models are not a desirable goal in themselves [AF92], but rather an unavoidable compromise to obtain acceptable performance and availability [AW94, Bre00, XSK+14]. These works try in general to minimize the violations of strong consistency, as these create anomalies for programmers and users. They further emphasize the low probability of such violations in their real deployments [DHJ+07].

In this deliverable we therefore present two contributions that aim to improve the state of the art of consistency criteria for large-scale geo-replicated systems. The first contribution, published in [FTR15], consists of a hybrid consistency criterion that links the strength of data consistency with the proximity of the participating nodes. The second, published in [PMJ15], addresses the limitations of eventual consistency by precisely defining the nature of the converged state.

Motivation and problem statement

Roadmap This deliverable consists of 7 sections. Section 2 introduces the definition of our first contribution, fisheye consistency. Then, Section 3 builds on top of this communication abstraction a distributed algorithm implementing this hybrid proximity-based data consistency condition. Section 4 introduces preliminary notions towards the definition of our second contribution. Section 5 defines the new update-consistency criterion. Section 6 presents a generic construction for any UQ-ADT object with a sequential specification. Finally, Section 7 concludes the document.

† We use indifferently participant or process to designate the computing entities that invoke the distributed object.

2 Fisheye Consistency

In spite of their benefits, the above consistency conditions generally ignore the relative "distance" between nodes in the underlying "infrastructure", where the notions of "distance" and "infrastructure" may be logical or physical, depending on the application. This is unfortunate as distributed systems must scale out and geo-replication is becoming more common. In a geo-replicated system, the network latency and bandwidth connecting nearby servers is usually at least an order of magnitude better than what is obtained between remote servers. This means that the cost of maintaining strong consistency among nearby nodes becomes affordable compared to the overall network costs and latencies in the system.

Some production-grade systems acknowledge the importance of distance when enforcing consistency, and do propose consistency mechanisms based on node locations in a distributed system (e.g. whether nodes are located in the same or in different data-centers). Unfortunately these production-grade systems usually do not distinguish between semantics and implementation. Rather, their consistency model is defined in operational terms, whose full implications can be difficult to grasp. In Cassandra [LM10], for instance, the application can specify for each operation the type of consistency guarantee it desires. For example, the constraints QUORUM and ALL require the involvement of a quorum of replicas and of all replicas, respectively; while LOCAL_QUORUM is satisfied when a quorum of the local data center is contacted, and EACH_QUORUM requires a quorum in each data center. These guarantees are defined by their implementation, but do not provide the programmer with a precise image of the consistency they deliver.

The need to take into account "distance" into consistency models, and the current lack of any formal underpinning to do so are exactly what motivates the hybridization of consistency conditions that we propose in our work (which we call *fisheye consistency*). Fisheye consistency conditions provide strong guarantees only for operations issued at nearby servers. In particular, there are many applications where one can expect that concurrent operations on the same objects are likely to be generated by geographically nearby nodes, e.g., due to business hours in different time zones, or because these objects represent localized information, etc. In such situations, a fisheye consistency condition would in fact provide global strong consistency at the cost of maintaining only locally strong consistency.

Consider for instance a node A that is "close" to a node B, but "far" from a node C, a causally consistent read/write register will offer the same (weak) guarantees to A on the operations of B, as on the operations of C. This may be suboptimal, as many applications could benefit from varying levels of consistency conditioned on "how far" nodes are from each other. Stated differently: a node can accept that "remote" changes only reach it with weak guarantees (e.g., because information takes time to travel), but it wants changes "close" to it to come with strong guarantees (as "local" changes might impact it more directly).

In this work, we propose to address this problem by integrating a notion of *node proximity* in the definition of *data consistency*. To that end, we formally define a new family of hybrid consistency models that links the strength of data consistency with the proximity of the participating nodes. In our approach, a particular hybrid model takes as input a proximity graph, and two consistency conditions, taken from a set of totally ordered consistency conditions, namely a strong one and a weaker one. A classical set of totally ordered conditions is the following one: linearizability, sequential consistency, causal consistency, and PRAM-consistency [LS88]. Moreover, as already said, the notion of proximity can be geographical (cluster-based physical distribution of the nodes), or purely logical (as in some peer-to-peer systems).

The philosophy we advocate is related to that of Parallel Snapshot Isolation (PSI) proposed in [SPAL11]. PSI combines strong consistency (Snapshot Isolation) for transactions started at nodes in the same site of a geo-replicated system, but only ensures causality among transactions started at different sites. In addition, PSI prevents write-write conflicts by preventing concurrent transactions with conflicting write sets, with the exception of commutable objects.

Although PSI and our work operate at different granularities (fisheye-consistency is expressed on individual operations, each accessing a single object, while PSI addresses general transactions), they both show the interest of consistency conditions in which nearby nodes enjoy stronger semantics than remote ones. In spite of this similitude, however, the family of consistency conditions we propose distinguishes itself from PSI in a number of key dimensions. First, PSI is a specific condition while fisheye-consistency offers a general framework for defining multiple such conditions. PSI only distinguished between nodes at the same physical site and remote nodes, whereas fisheye-consistency accepts arbitrary proximity graphs, which can be physical or logical. Finally, the definition of PSI is given in [SPAL11] by a reference implementation, whereas fisheye-consistency is defined in functional terms as restrictions on the ordering of operations that can be seen by applications, independently of the implementation we propose. As a result, we believe that our formalism makes it easier for users to express and understand the semantics of a given consistency condition and to prove the correctness of a program written w.r.t. such a condition.

2.1 System Model

The system consists of n processes denoted p_1, ..., p_n. We note Π the set of all processes. Each process is sequential and asynchronous. "Asynchronous" means that each process proceeds at its own speed, which is arbitrary, may vary with time, and remains always unknown to the other processes. Said differently, there is no notion of a global time that could be used by the processes.

Processes communicate by sending and receiving messages through channels. Each channel is reliable (no message loss, duplication, creation, or corruption), and asynchronous (transit times are arbitrary but finite, and remain unknown to the processes). Each pair of processes is connected by a bi-directional channel.

2.1.1 Basic notions and definitions

This section is a short reminder of the fundamental notions typically used to define the consistency guarantees of distributed objects, namely, operation, history, partial order on operations, and history equivalence. Interested readers will find in-depth presentations of these notions in textbooks such as [AW04, HS08, Lyn96, Ray12].

Concurrent objects with sequential specification A concurrent object is an object that can be simultaneously accessed by different processes. At the application level the processes interact through concurrent objects [HS08, Ray12]. Each object is defined by a sequential specification, which is a set including all the correct sequences of operations and their results that can be applied to and obtained from the object. These sequences are called *legal* sequences.

Execution history The execution of a set of processes interacting through objects is captured by a *history* $\widehat{H} = (H, \rightarrow_H)$, where $\rightarrow_H$ is a partial order on the set H of the object operations invoked by the processes.

Concurrency and sequential history If two operations are not ordered in a history, they are said to be *concurrent*. A history is said to be *sequential* if it does not include any concurrent operations. In this case, the partial order $\rightarrow_H$ is a total order.

Equivalent history Let $\widehat{H}|p$ represent the projection of $\widehat{H}$ onto the process p, i.e., the restriction of $\widehat{H}$ to operations occurring at process p. Two histories $\widehat{H}_1$ and $\widehat{H}_2$ are *equivalent* if no process can distinguish them, i.e., $\forall p \in \Pi : \widehat{H}_1|p = \widehat{H}_2|p$.

Legal history $\widehat{H}$ being a sequential history, let $\widehat{H}|X$ represent the projection of $\widehat{H}$ onto the object X. A history $\widehat{H}$ is *legal* if, for any object X, the sequence $\widehat{H}|X$ belongs to the specification of X.

Process Order Notice that since we assumed that processes are sequential, in the following, we restrict the discussion to execution histories $\widehat{H}$ for which for every process p, $\widehat{H}|p$ is sequential. This total order is also called the *process order* for p.

2.2 The Family of Fisheye Consistency Conditions

This section introduces a hybrid consistency model based on (a) two consistency conditions and (b) the notion of a proximity graph defined on the computing nodes (processes). The two consistency conditions must be totally ordered in the sense that any execution satisfying the stronger one also satisfies the weaker one. Linearizability and SC define such a pair of consistency conditions, and similarly SC and CC are such a pair.

2.2.1 The notion of a proximity graph

Let us assume that for physical or logical reasons linked to the application, each process (node) can be considered either close to or remote from other processes. This notion of "closeness" can be captured trough a *proximity graph* denoted $\mathcal{G} = (\Pi, E_{\mathcal{G}} \subseteq \Pi \times \Pi)$, whose vertices are the n processes of the system (Π). The edges are undirected. $N_{\mathcal{G}}(p_i)$ denotes the neighbors of p_i in $\mathcal{G}$.

The aim of $\mathcal{G}$ is to state the level of consistency imposed on processes in the following sense: the existence of an edge between two processes in $\mathcal{G}$ imposes a stronger data consistency level than between processes not connected in $\mathcal{G}$.

Example To illustrate the semantic of $\mathcal{G}$, we extend the original scenario that Ahamad, Niger *et al* use to motivate causal consistency in [ANB+95]. Consider the three processes of Figure 1, *paris*, *berlin*, and *new-york*. Processes *paris* and *berlin* interact closely with one another and behave symmetrically : they concurrently write the shared variable X, then set the flags R and S respectively to 1, and finally read X. By contrast, process *new-york* behaves sequentially w.r.t. *paris* and *berlin*: *new-york* waits for *paris* and *berlin* to write on X, using the flags R and S, and then writes X.

process *paris* **is**	**process** *berlin* **is**	**process** *new-york* **is**
$X \leftarrow 1$	$X \leftarrow 2$	**repeat** $c \leftarrow R$ **until** $c = 1$
$R \leftarrow 1$	$S \leftarrow 1$	**repeat** $d \leftarrow S$ **until** $d = 1$
$a \leftarrow X$	$b \leftarrow X$	$X \leftarrow 3$
end process	**end process**	**end process**

Figure 1: *new-york* does not need to be closely synchronized with *paris* and *berlin*, calling for a hybrid form of consistency

If we assume a model that provides causal consistency at a minimum, the write of X by *new-york* is guaranteed to be seen after the writes of *paris* and *berlin* by all processes (because *new-york* waits on R and S to be set to 1). Causal consistency however does not impose any consistent order on the writes of *paris* and *berlin* on X. In the execution shown on Figure 2, this means that although *paris* reads 2 in X (and thus sees the write of *berlin* after its own write), *berlin* might still read 1 in b (thus perceiving 'X.write(1)' and 'X.write(2)' in the opposite order to that of *paris*).

Sequential consistency removes this ambiguity: in this case, in Figure 2, *berlin* can only read 2 (the value it wrote) or 3 (written by *new-york*), but not 1. Sequential consistency is however too

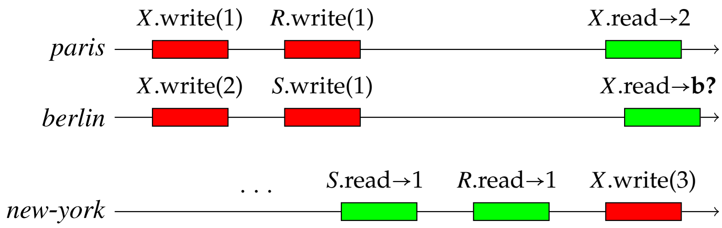


Figure 2: Executing the program of Figure 1.

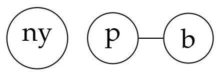


Figure 3: Capturing the synchronization needs of Fig. 1 with a proximity graph $\mathcal{G}$

strong here: because the write operation of *new-york* is already causally ordered with those of *paris* and *berlin*, this operation does not need any additional synchronization effort. This situation can be seen as an extension of the *write concurrency freedom* condition introduced in [ANB+95]: *new-york* is here free of concurrent write w.r.t. *paris* and *berlin*, making causal consistency equivalent to sequential consistency for *new-york*. *paris* and *berlin* however write to X concurrently, in which case causal consistency is not enough to ensure strongly consistent results.

If we assume *paris* and *berlin* execute in the same data center, while *new-york* is located on a distant site, this example illustrates a more general case in which, because of a program's logic or activity patterns, no operations at one site ever conflict with those at another. In such a situation, rather than enforce a strong (and costly) consistency in the whole system, we propose a form of consistency that is strong for processes within the same site (here *paris* and *berlin*), but weak between sites (here between *paris*, *berlin* on one hand and *new-york* on the other).

In our model, the synchronization needs of individual processes are captured by the *proximity graph* $\mathcal{G}$ introduced at the start of this section and shown in Figure 3: *paris* and *berlin* are connected, meaning the operations they execute should be perceived as strongly consistent w.r.t. one another ; *new-york* is neither connected to *paris* nor *berlin*, meaning a weaker consistency is allowed between the operations executed at *new-york* and those of *paris* and *berlin*.

2.2.2 Fisheye consistency for the pair (sequential consistency, causal consistency)

When applied to the scenario of Figure 2, fisheye consistency combines two consistency conditions (a strong and a weaker one, here causal and sequential consistency) and a proximity graph to form an hybrid distance-based consistency condition, which we call *$\mathcal{G}$-fisheye (SC,CC)-consistency*.

The intuition in combining SC and CC is to require that (write) operations be observed in the same order by all processes if:

- They are causally related (as in causal consistency),
- Or they occur on "close" nodes (as defined by $\mathcal{G}$).

Formal definition Formally, we say that a history $\widehat{H}$ is $\mathcal{G}$-fisheye (SC,CC)-consistent if:

- There is a causal order $\leadsto_H$ induced by $\widehat{H}$ (as in causal consistency); and

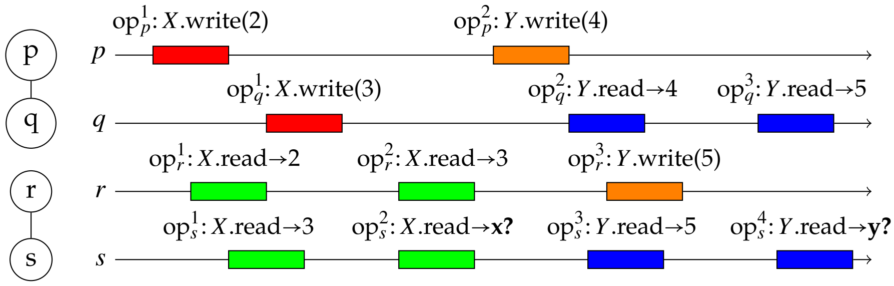


Figure 4: Illustrating $\mathcal{G}$-fisheye (SC,CC)-consistency

- $\rightsquigarrow_H$ can be extended to a subsuming order $\overset{\star}{\rightsquigarrow}_{H,\mathcal{G}}$ (i.e. $\rightsquigarrow_H \subseteq \overset{\star}{\rightsquigarrow}_{H,\mathcal{G}}$) so that

$$\forall p,q \in \mathcal{G} : (\overset{\star}{\rightsquigarrow}_{H,\mathcal{G}})|\{p,q\} \text{ is a total order}$$

  where $(\overset{\star}{\rightsquigarrow}_{H,\mathcal{G}})|(\{p,q\}\cap W)$ is the restriction of $\overset{\star}{\rightsquigarrow}_{H,\mathcal{G}}$ to the write operations of p and q; and

- for each process p_i there is a history $\widehat{S_i}$ that
  - (a) is sequential and legal;
  - (b) is equivalent to $\widehat{H}|(p_i+W)$; and
  - (c) respects $\overset{\star}{\rightsquigarrow}_{H,\mathcal{G}}$, i.e., $(\overset{\star}{\rightsquigarrow}_{H,\mathcal{G}})|(p_i+W) \subseteq (\rightarrow_{S_i})$.

If we apply this definition to the example of Figure 2 with the proximity graph proposed in Figure 3 we obtain the following: because *paris* and *berlin* are connected in $\mathcal{G}$, X.write(1) by *paris* and X.write(2) by *berlin* must be totally ordered in $\overset{\star}{\rightsquigarrow}_{H,\mathcal{G}}$ (and hence in any sequential history $\widehat{S_i}$ perceived by any process p_i). X.write(3) by *new-york* must be ordered after the writes on X by *paris* and *berlin* because of the causality imposed by $\rightsquigarrow_H$. As a result, if the system is $\mathcal{G}$-fisheye (SC,CC)-consistent, **b?** can be equal to 2 or 3, but not to 1. This set of possible values is as in sequential consistency, with the difference that $\mathcal{G}$-fisheye (SC,CC)-consistency does not impose any total order on the operation of *new-york*.

Given a system of n processes, let $\varnothing$ denote the graph $\mathcal{G}$ with no edges, and K denote the graph $\mathcal{G}$ with an edge connecting each pair of distinct processes. It is easy to see that CC is $\varnothing$-fisheye (SC,CC)-consistency. Similarly SC is K-fisheye (SC,CC)-consistency.

A larger example Figure 4 and Table 1 illustrate the semantic of $\mathcal{G}$-fisheye (SC,CC) consistency on a second, larger, example. In this example, the processes p and q on one hand, and r and s on the other hand, are neighbors in the proximity graph $\mathcal{G}$ (shown on the left). There are two pairs of write operations: op_p^1 and op_q^1 on the register X, and op_p^2 and op_r^3 on the register Y. In a sequentially consistency history, both pairs of writes must be seen in the same order by all processes. As a consequence, if r sees the value 2 first (op_r^1) and then the value 3 (op_r^2) for X, s must do the same, and only the value 3 can be returned by **x?**. For the same reason, only the value 3 can be returned by **y?**, as shown in the first line of Table 1.

In a causally consistent history, however, both pairs of writes ($\{\mathrm{op}_p^1, \mathrm{op}_q^1\}$ and $\{\mathrm{op}_p^2, \mathrm{op}_r^3\}$) are causally independent. As a result, any two processes can see each pair in different orders. **x?** may return 2 or 3, and **y?** 4 or 5 (second line of Table 1).

$\mathcal{G}$-fisheye (SC,CC)-consistency provides intermediate guarantees: because p and q are neighbors in $\mathcal{G}$, op_p^1 and op_q^1 must be observed in the same order by all processes. **x?** must return 3, as in a sequentially consistent history. However, because p and r are not connected in $\mathcal{G}$, op_p^2 and op_r^3

Table 1: Possible executions for the history of Figure 4

Consistency	**x?**	**y?**
Sequential Consistency	3	5
Causal Consistency	{2,3}	{4,5}
$\mathcal{G}$-fisheye (SC,CC)-consistency	3	{4,5}

may be seen in different orders by different processes (as in a causally consistent history), and **y?** may return 4 or 5 (last line of Table 1).

3 Implementing Fisheye Consistency

Our implementation of $\mathcal{G}$-fisheye (SC,CC)-consistency relies on a broadcast operation with hybrid ordering guarantees. In this section, we present this hybrid broadcast abstraction, before moving on the actual implementation of of $\mathcal{G}$-fisheye (SC,CC)-consistency in Section 3.3.

3.1 $\mathcal{G}$-fisheye (SC,CC)-broadcast: definition

The hybrid broadcast we proposed, denoted $\mathcal{G}$-(SC,CC)-broadcast, is parametrized by a proximity graph $\mathcal{G}$ which determines which kind of delivery order should be applied to which messages, according to the position of the sender in the graph $\mathcal{G}$. Messages (SC,CC)-broadcast by processes which are neighbors in $\mathcal{G}$ must be delivered in the same order at all the processes, while the delivery of the other messages only need to respect causal order.

The (SC,CC)-broadcast abstraction provides the processes with two operations, denoted $\mathsf{TOCO\_broadcast}()$ and $\mathsf{TOCO\_deliver}()$. We say that messages are toco-broadcast and toco-delivered.

Causal message order. Let M be the set of messages that are toco-broadcast. The causal message delivery order, denoted $\leadsto_M$, is defined as follows [BJ87, RST91]. Let $m_1, m_2 \in M$; $m_1 \leadsto_M m_2$, iff one of the following conditions holds:

- m_1 and m_2 have been toco-broadcast by the same process, with m_1 first;
- m_1 was toco-delivered by a process p_i before this process toco-broadcast m_2;
- There exists a message m such that $(m_1 \leadsto_M m) \wedge (m \leadsto_M m_2)$.

Definition of the $\mathcal{G}$-fisheye (SC,CC)-broadcast. The (SC,CC)-broadcast abstraction is defined by the following properties.

- *Validity.* If a process toco-delivers a message m, this message was toco-broadcast by some process. (No spurious message.)
- *Integrity.* A message is toco-delivered at most once. (No duplication.)
- *$\mathcal{G}$-delivery order.* For all the processes p and q such that (p,q) is an edge of $\mathcal{G}$, and for all the messages m_p and m_q such that m_p was toco-broadcast by p and m_q was toco-broadcast by q, if a process toco-delivers m_p before m_q, no process toco-delivers m_q before m_p.
- *Causal order.* If $m_1 \leadsto_M m_2$, no process toco-delivers m_2 before m_1.

- *Termination.* If a process toco-broadcasts a message m, this message is toco-delivered by all processes.

It is easy to see that if $\mathcal{G}$ has no edges, this definition boils down to causal delivery, and if $\mathcal{G}$ is fully connected (clique), this definition specifies total order delivery respecting causal order. Finally, if $\mathcal{G}$ is fully connected and we suppress the "causal order" property, the definition boils to total order delivery.

3.2 $\mathcal{G}$-fisheye (SC,CC)-broadcast: algorithm

3.2.1 Local variables.

To implement the $\mathcal{G}$-fisheye (SC,CC)-broadcast abstraction, each process p_i manages three local variables.

- $causal_i[1..n]$ is a local vector clock used to ensure a causal delivery order of the messages; $causal_i[j]$ is the sequence number of the next message that p_i will toco-deliver from p_j.
- $total_i[1..n]$ is a vector of logical clock values such that $total_i[i]$ is the local logical clock of p_i (Lamport's clock), and $total_i[j]$ is the value of $total_j[j]$ as known by p_i.
- $pending_i$ is a set containing the messages received and not yet toco-delivered by p_i.

3.2.2 Description of the algorithm.

Let us remind that for simplicity, we assume that the channels are FIFO. Algorithm 1 describes the behavior of a process p_i. This behavior is decomposed into four parts.

The first part (lines 1-6) is the code of the operation TOCO_broadcast(m). Process p_i first increases its local clock $total_i[i]$ and sends the protocol message $\text{TOCOBC}(m, \langle causal_i[\cdot], total_i[i], i\rangle)$ to each other process. In addition to the application message m, this protocol message carries the control information needed to ensure the correct toco-delivery of m, namely, the local causality vector ($causal_i[1..n]$), and the value of the local clock ($total_i[i]$). Then, this protocol message is added to the set $pending_i$ and $causal_i[i]$ is increased by 1 (this captures the fact that the future application messages toco-broadcast by p_i will causally depend on m).

The second part (lines 7-14) is the code executed by p_i when it receives a protocol message $\text{TOCOBC}(m, \langle s\_caus_j^m[\cdot],\ s\_tot_j^m,\ j\rangle)$ from p_j. When this occurs p_i adds first this protocol message to $pending_i$, and updates its view of the local clock of p_j ($total_i[j]$) to the sending date of the protocol message (namely, $s\_tot_j^m$). Then, if the local clock of p_i is late ($total_i[i] \leq s\_tot_j^m$), p_i catches up (line 11), and informs the other processes of it (line 12).

The third part (lines 15-17) is the processing of a catch up message from a process p_j. In this case, p_i updates its view of p_j's local clock to the date carried by the catch up message. Let us notice that, as channels are FIFO, a view $stotal_i[j]$ can only increase.

The final part (lines 18-31) is a background task executed by p_i, where the application messages are toco-delivered. The set C contains the protocol messages that were received, have not yet been toco-delivered, and are "minimal" with respect to the causality relation $\leadsto_M$. This minimality is determined from the vector clock $s\_caus_j^m[1..n]$, and the current value of p_i's vector clock ($causal_i[1..n]$). If only causal consistency was considered, the messages in C could be delivered.

Then, p_i extracts from C the messages that can be toco-delivered. Those are usually called *stable* messages. The notion of stability refers here to the delivery constraint imposed by the proximity graph $\mathcal{G}$. More precisely, a set T_1 is first computed, which contains the messages of C that (thanks

Algorithm 1 The $\mathcal{G}$-fisheye (SC,CC)-broadcast algorithm executed by p_i

1: **operation** TOCO_broadcast(m)
2: $total_i[i] \leftarrow total_i[i] + 1$
3: **for all** $p_j \in \Pi \setminus \{p_i\}$ **do** send TOCOBC$(m, \langle causal_i[\cdot], total_i[i], i\rangle)$ to p_j
4: $pending_i \leftarrow pending_i \cup \big\langle m, \langle causal_i[\cdot], total_i[i], i\rangle\big\rangle$
5: $causal_i[i] \leftarrow causal_i[i] + 1$
6: **end operation**

7: **on receiving** TOCOBC$(m, \langle s_caus_j^m[\cdot], s_tot_j^m, j\rangle)$
8: $pending_i \leftarrow pending_i \cup \big\langle m, \langle s\_caus_j^m[\cdot], s\_tot_j^m, j\rangle\big\rangle$
9: $total_i[j] \leftarrow s\_tot_j^m$ ▷ Last message from p_j had timestamp $s\_tot_j^m$
10: **if** $total_i[i] \le s\_tot_j^m$ **then**
11: $total_i[i] \leftarrow s\_tot_j^m + 1$ ▷ Ensuring global logical clocks
12: **for all** $p_k \in \Pi \setminus \{p_i\}$ **do** send CATCH_UP$(total_i[i], i)$ to p_k
13: **end if**
14: **end on receiving**

15: **on receiving** CATCH_UP$(last_date_j, j)$
16: $total_i[j] \leftarrow last\_date_j$
17: **end on receiving**

18: **background task** T **is**
19: **loop forever**
20: **wait until** $C \neq \emptyset$ **where**
21: $C \equiv \left\{ \big\langle m, \langle s\_caus_j^m[\cdot], s\_tot_j^m, j\rangle\big\rangle \in pending_i \;\middle|\; s\_caus_j^m[\cdot] \le causal_i[\cdot] \right\}$
22: **wait until** $T_1 \neq \emptyset$ **where**
23: $T_1 \equiv \left\{ \big\langle m, \langle s\_caus_j^m[\cdot], s\_tot_j^m, j\rangle\big\rangle \in C \;\middle|\; \forall p_k \in N_{\mathcal{G}}(p_j) : \langle total_i[k], k\rangle > \langle s\_tot_j^m, j\rangle \right\}$
24: **wait until** $T_2 \neq \emptyset$ **where**
25: $T_2 \equiv \left\{ \big\langle m, \langle s\_caus_j^m[\cdot], s\_tot_j^m, j\rangle\big\rangle \in T_1 \;\middle|\; \begin{array}{r} \forall p_k \in N_{\mathcal{G}}(p_j), \\ \forall \big\langle m_k, \langle s\_caus_k^{m_k}[\cdot], s\_tot_k^{m_k}, k\rangle\big\rangle \\ \in pending_i : \\ \langle s\_tot_k^{m_k}, k\rangle > \langle s\_tot_j^m, j\rangle \end{array} \right\}$
26: $\big\langle m_0, \langle s\_caus_{j_0}^{m_0}[\cdot], s\_tot_{j_0}^{m_0}, j_0\rangle\big\rangle \leftarrow \underset{\langle m, \langle s\_caus_j^m[\cdot], s\_tot_j^m, j\rangle\rangle \in T_2}{\arg\min} \big\{\langle s\_tot_j^m, j\rangle\big\}$
27: $pending_i \leftarrow pending_i \setminus \big\langle m_0, \langle s\_caus_{j_0}^{m_0}[\cdot], s\_tot_j^m, j_0\rangle\big\rangle$
28: TOCO_deliver(m_0) to application layer
29: **if** $j_0 \neq i$ **then** $causal_i[j_0] \leftarrow causal_i[j_0] + 1$ **end if** ▷ for $causal_i[i]$ see line 5
30: **end loop forever**
31: **end background task**

to the FIFO channels and the catch up messages) cannot be made unstable (with respect to the total delivery order defined by $\mathcal{G}$) by messages that p_i will receive in the future. Then the set T_2 is computed, which is the subset of T_1 such that no message received, and not yet toco-delivered, could make incorrect – w.r.t. $\mathcal{G}$ – the toco-delivery of a message of T_2.

Once a non-empty set T_2 has been computed, p_i extracts the message m whose timestamp $\langle s\_tot_j^m[j], j\rangle$ is "minimal" with respect to the timestamp-based total order (p_j is the sender of m). This message is then removed from $pending_i$ and toco-delivered. Finally, if $j \neq i$, $causal_i[j]$ is increased to take into account this toco-delivery (all the messages m' toco-broadcast by p_i in the future will be such that $m \rightsquigarrow m'$, and this is encoded in $causal_i[j]$). If $j = i$, this causality update was done at line 5.

Theorem 1. *Algorithm 1 implements a $\mathcal{G}$-fisheye (SC,CC)-broadcast.*

3.2.3 Proof of Theorem 1

The proof combines elements of the proofs of the traditional causal-order [BSS91, RST91] and total-order broadcast algorithms [Lam78, AW94] on which Algorithm 1 is based. It relies in particular on the monoticity of the clocks $causal_i[1..n]$ and $total_i[1..n]$, and the reliability and FIFO properties of the underlying communication channels. We first prove some useful lemmata, before proving termination, causal order, and $\mathcal{G}$-delivery order in intermediate theorems. We finally combine these intermediate results to prove Theorem 1.

We use the usual partial order on vector clocks:

$$C_1[\cdot] \leq C_2[\cdot] \text{ iff } \forall p_i \in \Pi : C_1[i] \leq C_2[i]$$

with its accompanying strict partial order:

$$C_1[\cdot] < C_2[\cdot] \text{ iff } C_1[\cdot] \leq C_2[\cdot] \wedge C_1[\cdot] \neq C_2[\cdot]$$

We use the lexicographic order on the scalar clocks $\langle s\_tot_j, j\rangle$:

$$\langle s\_tot_j, j\rangle < \langle s\_tot_i, i\rangle \text{ iff } (s\_tot_j < s\_tot_i) \vee (s\_tot_j = s\_tot_i \wedge i < j)$$

We start by three useful lemmata on $causal_i[\cdot]$ and $total_i[\cdot]$. These lemmata establish the traditional properties expected of logical and vector clocks.

Lemma 1. The following holds on the clock values taken by $causal_i[\cdot]$:

1. The successive values taken by $causal_i[\cdot]$ in Process p_i are monotonically increasing.
2. The sequence of $causal_i[\cdot]$ values attached to TOCOBC messages sent out by Process p_i are strictly increasing.

Proof Proposition 1 is derived from the fact that the two lines that modify $causal_i[\cdot]$ (lines 5, and 29) only increase its value. Proposition 2 follows from Proposition 1 and the fact that line 5 insures successive TOCOBC messages cannot include identical $causal_i[i]$ values. $\square_{Lemma\ 1}$

Lemma 2. The following holds on the clock values taken by $total_i[\cdot]$:

1. The successive values taken by $total_i[i]$ in Process p_i are monotonically increasing.
2. The sequence of $total_i[i]$ values included in TOCOBC and CATCH_UP messages sent out by Process p_i are strictly increasing.

3. The successive values taken by $total_i[\cdot]$ in Process p_i are monotonically increasing.

Proof Proposition 1 is derived from the fact that the lines that modify $total_i[i]$ (lines 2 and 11) only increase its value (in the case of line 11 because of the condition at line 10). Proposition 2 follows from Proposition 1, and the fact that lines 2 and 11 insures successive TOBOBC and CATCH_UP messages cannot include identical $total_i[i]$ values.

To prove Proposition 3, we first show that:

$$\forall j \neq i: \text{ the successive values taken by } total_i[j] \text{ in } p_i \text{ are monotonically increasing.} \tag{1}$$

For $j \neq i$, $total_i[j]$ can only be modified at lines 9 and 16, by values included in TOBOBC and CATCH_UP messages, when these messages are received. Because the underlying channels are FIFO and reliable, Proposition 2 implies that the sequence of $last\_date_j$ and $s\_tot_j^m$ values received by p_i from p_j is also strictly increasing, which shows equation (1).

From equation (1) and Proposition 1, we conclude that the successive values taken by the vector $total_i[\cdot]$ in p_i are monotonically increasing (Proposition 3). $\square_{Lemma\ 2}$

Lemma 3. Consider an execution of the protocol. The following invariant holds: for $i \neq j$, if m is a message sent from p_j to p_i, then at any point of p_i's execution outside of lines 28-29, $s\_caus_j^m[j] < causal_i[j]$ iff that m has been toco-delivered by p_i.

Proof We first show that if m has been toco-delivered by p_i, then $s\_caus_j^m[j] < causal_i[j]$, outside of lines 28-29. This implication follows from the condition $s\_caus_j^m[\cdot] \leq causal_i[\cdot]$ at line 21, and the increment at line 29.

We prove the reverse implication by induction on the protocol's execution by process p_i. When p_i is initialized $causal_i[\cdot]$ is null:

$$causal_i^0[\cdot] = [0 \cdots 0] \tag{2}$$

because the above is true of any process, with Lemma 2, we also have

$$s\_caus_j^m[\cdot] \geq [0 \cdots 0] \tag{3}$$

for all message m that is toco-broadcast by Process p_j.

(2) and (3) imply that there are no messages sent by p_j so that $s\_caus_j^m[j] < causal_i^0[j]$, and the Lemma is thus true when p_i starts.

Let us now assume that the invariant holds at some point of the execution of p_i. The only step at which the invariant might become violated in when $causal_i[j_0]$ is modified for $j_0 \neq i$ at line 29. When this increment occurs, the condition $s\_caus_{j_0}^m[j_0] < causal_i[j_0]$ of the lemma potentially becomes true for additional messages. We want to show that there is only one single additional message, and that this message is m_0, the message that has just been delivered at line 28, thus completing the induction, and proving the lemma.

For clarity's sake, let us denote $causal_i^\circ[j_0]$ the value of $causal_i[j_0]$ just before line 29, and $causal_i^\bullet[j_0]$ the value just after. We have $causal_i^\bullet[j_0] = causal_i^\circ[j_0] + 1$.

We show that $s\_caus_{j_0}^{m_0}[j_o] = causal_i^\circ[j_0]$, where $s\_caus_{j_0}^{m_0}[\cdot]$ is the causal timestamp of the message m_0 delivered at line 28. Because m_0 is selected at line 26, this implies that $m_0 \in T_2 \subseteq T_1 \subseteq C$. Because $m_0 \in C$, we have

$$s\_caus_{j_0}^{m_0}[\cdot] \leq causal_i^\circ[\cdot] \tag{4}$$

at line 21, and hence

$$s\_caus_{j_0}^{m_0}[j_0] \leq causal_i^\circ[j_0] \tag{5}$$

At line 21, m_0 has not been yet delivered (otherwise it would not be in $pending_i$). Using the contrapositive of our induction hypothesis, we have

$$s\_caus_{j_0}^{m_0}[j_0] \geq causal_i^{\circ}[j_0] \tag{6}$$

(5) and (6) yield

$$s\_caus_{j_0}^{m_0}[j_0] = causal_i^{\circ}[j_0] \tag{7}$$

Because of line 5, m_0 is the only message tobo_broadcast by P_{j_0} whose causal timestamp verifies (7). From this unicity and (7), we conclude that after $causal_i[j_0]$ has been incremented at line 29, if a message m sent by P_{j_0} verifies $s\_caus_{j_0}^{m}[j_0] < causal_i^{\bullet}[j_0]$, then

- either $s\_caus_{j_0}^{m}[j_0] < causal_i^{\bullet}[j_0] - 1 = causal_i^{\circ}[j_0]$, and by induction assumption, m has already been delivered;
- or $s\_caus_{j_0}^{m}[j_0] = causal_i^{\bullet}[j_0] - 1 < causal_i^{\circ}[j_0]$, and $m = m_0$, and m has just been delivered at line 28.

$\square_{Lemma\ 3}$

Termination

Theorem 2. *All messages toco-broadcast using Algorithm 1 are eventually toco-delivered by all processes in the system.*

Proof We show Termination by contradiction. Assume a process p_i toco-broadcasts a message m_i with timestamp $\langle s\_caus_i^{m_i}[\cdot], s\_tot_i^{m_i}, i\rangle$, and that m_i is never toco-delivered by p_j.

If $i \neq j$, because the underlying communication channels are reliable, p_j receives at some point the TOCOBC message containing m_i (line 7), after which we have

$$\left\langle m_i, \langle s\_caus_i^{m_i}[\cdot], s\_tot_i^{m_i}, i\rangle \right\rangle \in pending_j \tag{8}$$

If $i = j$, m_i is inserted into $pending_i$ immediately after being toco-broadcast (line 4), and (8) also holds.

m_i might never be toco-delivered by p_j because it never meets the condition to be selected into the set C of p_j (noted C_j below) at line 21. We show by contradiction that this is not the case. First, and without loss of generality, we can choose m_i so that it has a minimal causal timestamp $s\_caus_i^{m_i}[\cdot]$ among all the messages that j never toco-delivers (be it from p_i or from any other process). Minimality means here that

$$\forall m_x, p_j \text{ never delivers } m_x \Rightarrow \neg(s\_caus_x^{m_x} < s\_caus_i^{m_i}) \tag{9}$$

Let us now assume m_i is never selected into C_j, i.e., we always have

$$\neg(s\_caus_i^{m_i}[\cdot] \leq causal_j[\cdot]) \tag{10}$$

This means there is a process p_k so that

$$s\_caus_i^{m_i}[k] > causal_j[k] \tag{11}$$

If $i = k$, we can consider the message m'_i sent by i just before m_i (which exists since the above implies $s\_caus_i^{m_i}[i] > 0$). We have $s\_caus_i^{m'_i}[i] = s\_caus_i^{m_i}[i] - 1$, and hence from (11) we have

$$s\_caus_i^{m'_i}[i] \geq causal_j[k] \tag{12}$$

Applying Lemma 3 to (12) implies that p_j never toco-delivers m'_i either, with $s\_caus_i^{m'_i}[i] < s\_caus_i^{m_i}[i]$ (by way of Proposition 2 of Lemma 1), which contradicts (9).

If $i \neq k$, applying Lemma 3 to $causal_i[\cdot]$ when p_i toco-broadcasts m_i at line 3, we find a message m_k sent by p_k with $s\_caus_k^{m_k}[k] = s\_caus_i^{m_i}[k] - 1$ such that m_k was received by p_i before p_i toco-broadcast m_i. In other words, m_k belongs to the causal past of m_i, and because of the condition on C (line 21) and the increment at line 29, we have

$$s\_caus_k^{m_k}[\cdot] < s\_caus_i^{m_i}[\cdot] \tag{13}$$

As for the case $i = k$, (11) also implies

$$s\_caus_k^{m_k}[k] \geq causal_j[k] \tag{14}$$

which with Lemma 3 implies that that p_j never delivers the message m_k from p_k, and with (13) contradicts m_i's minimality (9).

We conclude that if a message m_i from p_i is never toco-delivered by p_j, after some point m_i remains indefinitely in C_j

$$m_i \in C_j \tag{15}$$

Without loss of generality, we can now choose m_i with the smallest total order timestamp $\langle s\_tot_i^{m_i}, i\rangle$ among all the messages never delivered by p_j. Since these timestamps are totally ordered, and no timestamp is allocated twice, there is only one unique such message.

We first note that because channels are reliable, all processes $p_k \in N_{\mathcal{G}}(p_i)$ eventually receive the TOCOBC protocol message of p_i that contains m_i (line 7 and following). Lines 10-11 together with the monotonicity of $total_k[k]$ (Proposition 1 of Lemma 2), insure that at some point all processes p_k have a timestamp $total_k[k]$ strictly larger than $s\_tot_i^{m_i}$:

$$\forall p_k \in N_{\mathcal{G}}(p_i) : total_k[k] > s\_tot_i^{m_i} \tag{16}$$

Since all changes to $total_k[k]$ are systematically rebroadcast to the rest of the system using TOCOBC or CATCHUP protocol messages (lines 2 and 11), p_j will eventually update $total_j[k]$ with a value strictly higher than $s\_tot_i^{m_i}$. This update, together with the monotonicity of $total_j[\cdot]$ (Proposition 3 of Lemma 2), implies that after some point:

$$\forall p_k \in N_{\mathcal{G}}(p_i) : total_j[k] > s\_tot_i^{m_i} \tag{17}$$

and that m_i is selected in T_1^j. We now show by contradiction that m_i eventually progresses to T_2^j. Let us assume m_i never meets T_2^j's condition. This means that every time T_2^j is evaluated we have:

$$\begin{aligned}\exists p_k \in N_{\mathcal{G}}(p_i), \exists \big\langle m_k, \langle s\_caus_k^{m_k}[\cdot], s\_tot_k^{m_k}, k\rangle\big\rangle \in pending_j : \\ \langle s\_tot_k^{m_k}, k\rangle \leq \langle s\_tot_i^{m}, i\rangle\end{aligned} \tag{18}$$

Note that there could be different p_k and m_k satisfying (18) in each loop of Task T. However, because $N_{\mathcal{G}}(p_i)$ is finite, the number of timestamps $\langle s\_tot_k^{m_k}, k\rangle$ such that $\langle s\_tot_k^{m_k}, k\rangle \le \langle s\_tot_i^{m}, i\rangle$ is also finite. There is therefore one process p_{k_0} and one message m_{k_0} that appear infinitely often in the sequence of (p_k, m_k) that satisfy (18). Since m_{k_0} can only be inserted once into $pending_j$, this means m_{k_0} remains indefinitely into T_2^j, and hence $pending_j$, and is never delivered. (18) and the fact that $i \neq k_0$ (because $p_i \notin N_{\mathcal{G}}(p_i)$) yields

$$\langle s\_tot_k^{m_{k_0}}, k_0\rangle < \langle s\_tot_i^{m}, i\rangle \tag{19}$$

which contradicts our assumption that m_i has the smallest total order timestamps $\langle s\_tot_i^{m_i}, i\rangle$ among all messages never delivered to p_j. We conclude that after some point m_i remains indefinitely into T_2^j.

$$m_i \in T_2^j \tag{20}$$

If we now assume m_i is never returned by $\arg\min$ at line 26, we can repeat a similar argument on the finite number of timestamps smaller than $\langle s\_tot_i^{m}, i\rangle$, and the fact that once they have been removed form $pending_j$ (line 27), messages are never inserted back, and find another message m_k with a strictly smaller time-stamp that p_j that is never delivered. The existence of m_k contradicts again our assumption on the minimality of m_i's timestamp $\langle s\_tot_i^{m}, i\rangle$ among undelivered messages.

This shows that m_i is eventually delivered, and ends our proof by contradiction. $\square_{Theorem\ 2}$

Causal Order We prove the causal order property by induction on the causal order relation $\rightsquigarrow_M$.

Lemma 4. Consider m_1 and m_2, two messages toco-broadcast by Process p_i, with m_1 toco-broadcast before m_2. If a process p_j toco-delivers m_2, then it must have toco-delivered m_1 before m_2.

Proof We first consider the order in which the messages were inserted into $pending_j$ (along with their causal timestamps $s\_caus_i^{m_{1|2}}$). For $i = j$, m_1 was inserted before m_2 at line 4 by assumption. For $i \neq j$, we note that if p_j delivers m_2 at line 28, then m_2 was received from p_i at line 7 at some earlier point. Because channels are FIFO, this also means

$$m_1 \text{ was received and added to } pending_j \text{ before } m_2 \text{ was.} \tag{21}$$

We now want to show that when m_2 is delivered by p_j, m_1 is no longer in $pending_j$, which will show that m_1 has been delivered before m_2. We use an argument by contradiction. Let us assume that

$$\big\langle m_1, \langle s\_caus_i^{m_1}, s\_tot_i^{m_1}, i\rangle\big\rangle \in pending_j \tag{22}$$

at the start of the iteration of Task T which delivers m_2 to p_j. From Proposition 2 of Lemma 1, we have

$$s\_caus_i^{m_1} < s\_caus_i^{m_2} \tag{23}$$

which implies that m_1 is selected into C along with m_2 (line 21):

$$\big\langle m_1, \langle s\_caus_i^{m_1}, s\_tot_i^{m_1}, i\rangle\big\rangle \in C$$

Similarly, from Proposition 2 of Lemma 2 we have:

$$s\_tot_i^{m_1} < s\_tot_i^{m_2} \tag{24}$$

which implies that m_1 must also belong to T_1 and T_2 (lines 23 and 25). (24) further implies that $\langle s\_tot_i^{m_2}, i\rangle$ is not the minimal s_tot timestamp of T_2, and therefore $m_0 \neq m_2$ in this iteration of Task T. This contradicts our assumption that m_2 was delivered in this iteration; shows that (22) must be false; and therefore with (21) that m_1 was delivered before m_2. $\square_{Lemma\ 4}$

Lemma 5. Consider m_1 and m_2 so that m_1 was toco-delivered by a process p_i before p_i toco-broadcasts m_2. If a process p_j toco-delivers m_2, then it must have toco-delivered m_1 before m_2.

Proof Let us note p_k the process that has toco-broadcast m_1. Because m_2 is toco-broadcasts by p_i after p_i toco-delivers m_1 and increments $causal_i[k]$ at line 29, we have, using Lemma 3 and Proposition 1 of Lemma 1:

$$s\_caus_k^{m_1}[k] < s\_caus_i^{m_2}[k] \tag{25}$$

Because of the condition on set C at line 21, when p_j toco-delivers m_2 at line 28, we further have

$$s\_caus_i^{m_2}[\cdot] \leq causal_j[\cdot] \tag{26}$$

and hence using (25)

$$s\_caus_k^{m_1}[k] < s\_caus_i^{m_2}[k] \leq causal_j[k] \tag{27}$$

Applying Lemma 3 to (27), we conclude that $p_j$must have toco-delivered m_1 when it delivers m_2. $\square_{Lemma\ 5}$

Theorem 3. *Algorithm 1 respects causal order.*

Proof We finish the proof by induction on $\rightsquigarrow_M$. Let's consider three messages m_1, m_2, m_3 such that

$$m_1 \rightsquigarrow_M m_3 \rightsquigarrow_M m_2 \tag{28}$$

and such that:

- if a process toco-delivers m_3, it must have toco-delivered m_1;
- if a process toco-delivers m_2, it must have toco-delivered m_3;

We want to show that if a process toco-delivers m_2, it must have tolo-delivered m_1. This follows from the transitivity of temporal order. This result together with Lemmas 4 and 5 concludes the proof. $\square_{Theorem\ 3}$

$\mathcal{G}$-delivery order

Theorem 4. *Algorithm 1 respects $\mathcal{G}$-delivery order.*

Proof Let's consider four processes p_l, p_h, p_i, and p_j. p_l and p_h are connected in $\mathcal{G}$. p_l has toco-broadcast a message m_l, and p_h has toco-broadcast a message m_h. p_i has toco-delivered m_l before m_h. p_j has toco-delivered m_h. We want to show that p_j has toco-delivered m_l before m_h.

We first show that:

$$\langle s\_tot_h^{m_h}, h\rangle > \langle s\_tot_l^{m_l}, l\rangle \tag{29}$$

We do so by considering the iteration of the background task T (lines 18-18) of p_i that toco-delivers m_l. Because $p_h \in N_{\mathcal{G}}(p_l)$, we have

$$\langle total_i[h], h\rangle > \langle s\_tot_l^{m_l}, l\rangle \tag{30}$$

at line 23.

If m_h has not been received by p_i yet, then because of Lemma 3.2, and because communication channels are FIFO and reliable, we have:

$$\langle s\_tot_h^{m_h}, l\rangle > \langle total_i[h], h\rangle \tag{31}$$

which with (30) yields (29).

If m_h has already been received by p_i, by assumption it has not been toco-delivered yet, and is therefore in $pending_i$. More precisely we have:

$$\big\langle m_h, \langle s\_caus_h^{m_h}[\cdot], s\_tot_h^{m_h}, h\rangle\big\rangle \in pending_i \tag{32}$$

which, with $p_h \in N_{\mathcal{G}}(p_l)$, and the fact that m_l is selected in T_2^i at line 25 also gives us (29).

We now want to show that p_j must have toco-delivered m_l before m_h. The reasoning is somewhat the symmetric of what we have done. We consider the iteration of the background task T of p_j that toco-delivers m_h. By the same reasoning as above we have

$$\langle total_j[l], l\rangle > \langle s\_tot_h^{m_h}, h\rangle \tag{33}$$

at line 23.

Because of Lemma 3.2, and because communication channels are FIFO and reliable, (33) and (29) imply that m_l has already been received by p_j. Because m_h is selected in T_2^j at line 25, (29) implies that m_h is no longer in $pending_j$, and so must have been toco-delivered by p_j earlier, which concludes the proof. $\square_{Theorem\ 4}$

Theorem 1. *Algorithm 1 implements a $\mathcal{G}$-fisheye (SC,CC)-broadcast.*

Proof

- Validity and Integrity follow from the integrity and validity of the underlying communication channels, and from how a message m_j is only inserted once into $pending_i$ (at line 4 if $i = j$, at line 8 otherwise) and always removed from $pending_i$ at line 27 before it is toco-delivered by p_i at line 28;
- $\mathcal{G}$-delivery order follows from Theorem 4;
- Causal order follows from Theorem 3;
- Termination follows from Theorem 2.

$\square_{Theorem\ 1}$

3.3 An Algorithm Implementing $\mathcal{G}$-Fisheye (SC,CC)-Consistency

3.3.1 The high level object operations read and write

Algorithm 2 uses the $\mathcal{G}$-fisheye (SC,CC)-broadcast we have just presented to realized $\mathcal{G}$-fisheye (SC,CC)-consistency using a fast-read strategy. This algorithm is derived from the fast-read algorithm for sequential consistency proposed by Attiya and Welch [AW94], in which the total order broadcast has been replaced by our $\mathcal{G}$-fisheye (SC,CC)-broadcast.

The $\mathsf{write}(X, v)$ operation uses the $\mathcal{G}$-fisheye (SC,CC)-broadcast to propagate the new value of the variable X. To insure any other write operations that must be seen *before* $\mathsf{write}(X, v)$ by p_i are

Algorithm 2 Implementing $\mathcal{G}$-fisheye (SC,CC)-consistency, executed by p_i

```
1: operation X.write(v)
2:     TOCO_broadcast(WRITE(X, v, i))
3:     delivered_i ← false ;
4:     wait until delivered_i = true
5: end operation

6: operation X.read()
7:     return v_x
8: end operation

9: on toco_deliver WRITE(X, v, j)
10:    v_x ← v ;
11:    if (i = j) then delivered_i ← true endif
12: end on toco_deliver
```

properly processed, p_i enters a waiting loop (line 4), which ends after the message $\text{WRITE}(X,v,i)$ that has been toco-broadcast at line 2 is toco-delivered at line 11.

The read(X) operation simply returns the local copy v_x of X. These local copies are updated in the background when $\text{WRITE}(X,v,j)$ messages are toco-delivered.

Theorem 5. *Algorithm 2 implements $\mathcal{G}$-fisheye (SC,CC)-consistency.*

3.3.2 Proof of Theorem 5

The proof uses the causal order on messages $\rightsquigarrow_M$ provided by the $\mathcal{G}$-fisheye (SC,CC)-broadcast to construct the causal order on operations $\rightsquigarrow_H$. It then gradually extends $\rightsquigarrow_H$ to obtain $\overset{\star}{\rightsquigarrow}_{H,\mathcal{G}}$. It first uses the property of the broadcast algorithm on messages to-broadcast by processes that are neighbors in $\mathcal{G}$, and then adapts the technique used in [MZR95, Ray13] to show that WW (write-write) histories are sequentially consistent. The individual histories $\widehat{S_i}$ are obtained by taking a topological sort of $(\overset{\star}{\rightsquigarrow}_{H,\mathcal{G}})|(p_i + W)$.

For readability, we denote in the following $r_p(X,v)$ the read operation invoked by process p on object X that returns a value v (X.read $\to v$), and $w_p(X,v)$ the write operation of value v on object X invoked by process p (X.write(v)). We may omit the name of the process when not needed.

Let us consider a history $\widehat{H} = (H, \overset{po}{\to}_H)$ that captures an execution of Algorithm 2, i.e., $\overset{po}{\to}_H$ captures the sequence of operations in each process (process order, *po* for short). We construct the causal order $\rightsquigarrow_H$ required by the definition of Section 2.2.2 in the following, classical, manner:

- We connect each read operation $r_p(X,v) = X.\text{read} \to v$ invoked by process p (with $v \neq \bot$, the initial value) to the write operation $w(X,v) = X.\text{write}(v)$ that generated the $\text{WRITE}(X,v)$ message carrying the value v to p (line 10 in Algorithm 2). In other words, we add an edge $\langle w(X,v) \overset{rf}{\to} r_p(X,v)\rangle$ to $\overset{po}{\to}_H$ (with w and r_p as described above) for each read operation $r_p(X,v) \in H$ that does not return the initial value $\bot$. We connect initial read operations $r(X,\bot)$ to an $\bot$ element that we add to H.

  We call these additional relations *read-from links* (noted $\overset{rf}{\to}$).

- We take $\rightsquigarrow_H$ to be the transitive closure of the resulting relation.

$\rightsquigarrow_H$ is acyclic, as assuming otherwise would imply at least one of the $\text{WRITE}(X,v)$ messages was received before it was sent. $\rightsquigarrow_H$ is therefore an order. We now need to show $\rightsquigarrow_H$ is a *causal* *order*, i.e., that the result of each read operation $r(X,v)$ is the value of the latest write $w(X,v)$ that occurred before $r(X,v)$ in $\leadsto_H$ (said differently, that no read returns an overwritten value).

Lemma 6. $\leadsto_H$ is a causal order.

Proof We show this by contradiction. We assume without loss of generality that all values written are distinct. Let us consider $w_p(X,v)$ and $r_q(X,v)$ so that $w_p(X,v) \xrightarrow{rf} r_q(X,v)$, which implies $w_p(X,v) \leadsto_H r_q(X,v)$. Let us assume there exists a second write operation $w_r(X,v') \neq w_p(X,v)$ on the same object, so that

$$w_p(X,v) \leadsto_H w_r(X,v') \leadsto_H r_q(X,v) \tag{34}$$

(illustrated in Figure 5). $w_p(X,v) \leadsto_H w_r(X,v')$ means we can find a sequence of operations $op_i \in H$ so that

$$w_p(X,v) \rightarrow_0 op_0 \ldots \rightarrow_i op_i \rightarrow_{i+1} \ldots \rightarrow_k w_r(X,v') \tag{35}$$

with $\rightarrow_i \in \{\xrightarrow{po}_H, \xrightarrow{rf}\}, \forall i \in [1,k]$. The semantics of $\xrightarrow{po}_H$ and $\xrightarrow{rf}$ means we can construct a sequence of causally related (SC,CC)-broadcast messages $m_i \in M$ between the messages that are toco-broadcast by the operations $w_p(X,v)$ and $w_r(X,v')$, which we note $\text{WRITE}_p(X,v)$ and $\text{WRITE}_r(X,v')$ respectively:

$$\text{WRITE}_p(X,v) = m_0 \leadsto_M m_1 \ldots \leadsto_M m_i \leadsto_M \ldots \leadsto_M m_{k'} = \text{WRITE}_r(X,v') \tag{36}$$

where $\leadsto_M$ is the message causal order introduced in Section 3.1. We conclude that $\text{WRITE}_p(X,v) \leadsto_M \text{WRITE}_r(X,v')$, i.e., that $\text{WRITE}_p(X,v)$ belongs to the causal past of $\text{WRITE}_r(X,v')$, and hence that q in Figure 5 toco-delivers $\text{WRITE}_r(X,v')$ after $\text{WRITE}_p(X,v)$.

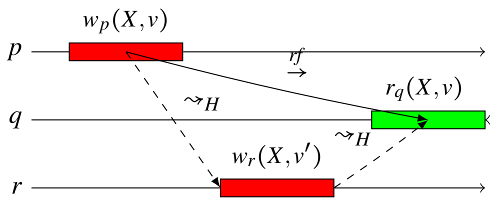


Figure 5: Proving that $\leadsto_H$ is causal by contradiction

We now want to show that $\text{WRITE}_r(X,v')$ is toco-delivered by q before q executes $r_q(X,v)$. We can apply the same reasoning as above to $w_r(X,v') \leadsto_H r_q(X,v)$, yielding another sequence of operations $op'_i \in H$:

$$w_r(X,v') \rightarrow'_0 op'_0 \ldots \rightarrow'_i op'_i \rightarrow'_{i+1} \ldots \rightarrow'_{k''} r_q(X,v) \tag{37}$$

with $\rightarrow'_i \in \{\xrightarrow{po}_H, \xrightarrow{rf}\}$. Because $r_q(X,v)$ does not generate any (SC,CC)-broadcast message, we need to distinguish the case where all op'_i relations correspond to the process order $\xrightarrow{po}_H$ (i.e., $op'_i = \xrightarrow{po}_H, \forall i$). In this case, $r = q$, and the blocking behavior of $X.\mathsf{write}()$ (line 4 of Algorithm 2), insures that $\text{WRITE}_r(X,v')$ is toco-delivered by q before executing $r_q(X,v)$. If at least one op'_i corresponds to the read-from relation, we can consider the latest one in the sequence, which will denote the toco-delivery of a $\text{WRITE}_z(Y,w)$ message by q, with $\text{WRITE}_r(X,v') \leadsto_M \text{WRITE}_z(Y,w)$. From the causality of the (SC,CC)-broadcast, we also conclude that $\text{WRITE}_r(X,v')$ is toco-delivered by q before executing $r_q(X,v)$.

Because q toco-delivers $\text{WRITE}_p(X,v)$ before $\text{WRITE}_r(X,v')$, and toco-delivers $\text{WRITE}_r(X,v')$ before it executes $r_q(X,v)$, we conclude that the value v of v_x is overwritten by v' at line 10 of Algorithm 2, and that $r_q(X,v)$ does not return v, contradicting our assumption that $w_p(X,v) \xrightarrow{rf} r_q(X,v)$, and concluding our proof that $\leadsto_H$ is a causal order. $\square_{Lemma\ 6}$

To construct $\overset{\star}{\rightsquigarrow}_{H,\mathcal{G}}$, as required by the definition of (SC,CC)-consistency (Section 2.2.2), we need to order the write operations of neighboring processes in the proximity graph $\mathcal{G}$. We do so as follows:

- We add an edge $w_p(X,v) \overset{ww}{\to} w_q(Y,w)$ to $\rightsquigarrow_H$ for each pair of write operations $w_p(X,v)$ and $w_q(Y,w)$ in H such that:
  - $(p,q) \in E_{\mathcal{G}}$ (i.e., p and q are connected in $\mathcal{G}$);
  - $w_p(X,v)$ and $w_q(Y,w)$ are not ordered in $\rightsquigarrow_H$;
  - The broadcast message $\text{WRITE}_p(X,v)$ of $w_p(X,v)$ has been toco-delivered before the broadcast message $\text{WRITE}_p(Y,w)$ of $w_q(Y,w)$ by all processes.

  We call these additional edges *ww links* (noted $\overset{ww}{\to}$).

- We take $\overset{\star}{\rightsquigarrow}_{H,\mathcal{G}}$ to be the recursive closure of the relation we obtain.

$\overset{\star}{\rightsquigarrow}_{H,\mathcal{G}}$ is acyclic, as assuming otherwise would imply that the underlying (SC,CC)-broadcast violates causality. Because of the $\mathcal{G}$-delivery order and termination of the toco-broadcast (Section 3.1), we know all pairs of $\text{WRITE}_p(X,v)$ and $\text{WRITE}_p(Y,w)$ messages with $(p,q) \in E_{\mathcal{G}}$ as defined above are toco-delivered in the same order by all processes. This insures that all write operations of neighboring processes in $\mathcal{G}$ are ordered in $\overset{\star}{\rightsquigarrow}_{H,\mathcal{G}}$.

We need to show that $\overset{\star}{\rightsquigarrow}_{H,\mathcal{G}}$ remains a causal order, i.e., that no read in $\overset{\star}{\rightsquigarrow}_{H,\mathcal{G}}$ returns an overwritten value.

Lemma 7. $\overset{\star}{\rightsquigarrow}_{H,\mathcal{G}}$ is a causal order.

Proof We extend the original causal order $\rightsquigarrow_M$ on the messages of an (SC,CC)-broadcast execution with the following order $\rightsquigarrow_M^{\mathcal{G}}$:

$m_1 \rightsquigarrow_M^{\mathcal{G}} m_2$ if

- $m_1 \rightsquigarrow_M m_2$; or
- m_1 was sent by p, m_2 by q, $(p,q) \in E_{\mathcal{G}}$, and m_1 is toco-delivered before m_2 by all processes; or
- there exists a message m_3 so that $m_1 \rightsquigarrow_M^{\mathcal{G}} m_3$ and $m_3 \rightsquigarrow_M^{\mathcal{G}} m_2$.

$\rightsquigarrow_M^{\mathcal{G}}$ captures the order imposed by an execution of an (SC,CC)-broadcast on its messages. The proof is then identical to that of Lemma 6, except that we use the order $\rightsquigarrow_M^{\mathcal{G}}$, instead of $\rightsquigarrow_M$. $\square_{Lemma\ 7}$

Theorem 5. *Algorithm 2 implements $\mathcal{G}$-fisheye (SC,CC)-consistency.*

Proof The order $\overset{\star}{\rightsquigarrow}_{H,\mathcal{G}}$ we have just constructed fulfills the conditions required by the definition of $\mathcal{G}$-fisheye (SC,CC)-consistency (Section 2.2.2):

- by construction $\overset{\star}{\rightsquigarrow}_{H,\mathcal{G}}$ subsumes $\rightsquigarrow_H$ ($\rightsquigarrow_H \subseteq \overset{\star}{\rightsquigarrow}_{H,\mathcal{G}}$);
- also by construction $\overset{\star}{\rightsquigarrow}_{H,\mathcal{G}}$, any pair of write operations invoked by processes p,q that are neighbors in $\mathcal{G}$ are ordered in $\overset{\star}{\rightsquigarrow}_{H,\mathcal{G}}$; i.e., $(\overset{\star}{\rightsquigarrow}_{H,\mathcal{G}})|(\{p,q\} \cap W)$ is a total order.

To finish the proof, we choose, for each process p_i, $\widehat{S_i}$ as one of the topological sorts of $(\overset{\star}{\leadsto}_{H,\mathcal{G}})|(p_i + W)$, following the approach of [MZR95, Ray13]. $\widehat{S_i}$ is sequential by construction. Because $\overset{\star}{\leadsto}_{H,\mathcal{G}}$ is causal, $\widehat{S_i}$ is legal. Because $\overset{\star}{\leadsto}_{H,\mathcal{G}}$ respects $\overset{po}{\to}_H$, $\widehat{S_i}$ is equivalent to $\widehat{H}|(p_i + W)$. Finally, $\widehat{S_i}$ respects $(\overset{\star}{\leadsto}_{H,\mathcal{G}})|(p_i + W)$ by construction. $\Box_{Theorem\ 5}$

4 Towards Update Consistency

We now introduce the second contribution of this deliverable. This contribution follows the long quest of the (a) strongest consistency criterion (there may exist several incomparable criteria) implementable for different types of objects in an asynchronous system where all but one process may crash (wait-free systems [Her91]). A contribution of this line of work consists in proving that weak consistency criteria such as eventual consistency and causal consistency cannot be combined is such systems. In this second part of our work we therefore explore the enforcement of eventual consistency. The relevance of eventual consistency has been illustrated many times. It is used in practice in many large scale applications such as Amazon's Dynamo highly available key-value store [DHJ+07]. It has been widely studied and many algorithms have been proposed to implement eventually consistent shared object. Conflict-free replicated data types (CRDT) [SPBZ11] give sufficient conditions on the specification of objects so that they can be implemented. More specifically, if all the updates made on the object commute or if the reachable states of the object form a semi-lattice then the object has an eventually consistent implementation [SPBZ11]. Unfortunately, many useful objects are not CRDTs.

The limitations of eventual consistency led to the study of stronger criteria such as strong eventual consistency [SPB+11]. Indeed, eventual consistency requires the convergence towards a *common state* without specifying which states are legal. In order to prove the correctness of a program, it is necessary to fully specify which behaviors are accepted for an object. The meaning of an operation often depends on the context in which it is executed. The notion of *intention* is widely used to specify collaborative editing [SJZ+98, LZM00]. The intention of an operation not only depends on the operation and the state on which it is done, but also on the intentions of the concurrent operations. In another solution [BZP+12], it is claimed that, it is sufficient to specify what the concurrent execution of all pairs of non-commutative operations should give (e.g. an error state). This result, acceptable for the shared s et, cannot be extended to other more complicated objects. In this case, any partial order of updates can lead to a different result. This approach was formalized in [BGYZ14], where the concurrent specification of an object is defined as a function of partially ordered sets of updates to a consistent state leading to specifications as complicated as the implementations themselves. Moreover, a concurrent specification of an object uses the notion of *concurrent events*. In message-passing systems, two events are concurrent if they are produced by different processes and each process produced its event before it received the notification message from the other process. In other words, the notion of concurrency depends on the implementation of an object not on its specification. Consequently, the final user may not know if two events are concurrent without explicitly tracking the underlying messages. A specification should be independent of the system on which it is implemented.

To avoid restricting our results to a given data structure, we first define a class of data types called UQ-ADT for *update-query abstract data type*. This class encompasses all data structures where an operation either modifies the state of the object (update) or returns a function on the current state of the object (query). This class excludes data types such as a stack where the pop operation removes the top of the stack and returns it (update and query at the same time). However, such operations can always be separated into a query and an update (lookup_top and delete_top in the case of the stack) which is not a problem as, in weak consistency models, it is impossible to ensure atomicity anyway. Based on this notion, we then present three main contributions.

- We prove that in a wait-free asynchronous system, it is not possible to implement eventual and causal consistency for all UQ-ADTs.
- We introduce *update consistency*, a new consistency criterion stronger than eventual consistency and for which the converging state must be consistent with a linearization of the updates.
- Finally, we prove that for any UQ-ADT object with a sequential specification there exists an update consistent implementation by providing a generic construction.

4.1 Abstract Data Types and Consistency Criteria

Before introducing the new consistency criterion, this section formalizes the notion of object and how a consistency criterion is defined. In distributed systems, sharing objects is a way to abstract message-passing communication between processes. The abstract type of these objects has a sequential specification, which we describe by means of a transition system that characterizes the sequential histories allowed for this object. However, shared objects are implemented in a distributed system using replication and the events of the distributed history generated by the execution of a distributed program is a partial order [Lam78]. The consistency criterion makes the link between the sequential specification of an object and a distributed execution that invokes it. This is done by characterizing the partially ordered histories of the distributed program that are acceptable. The formalization used in this deliverable is explained with more details in [PPJM14].

An abstract data type is specified using a transition system very close to Mealy machines [Mea55] except that infinite transition systems are allowed as many objects have an unbounded specification. As stated above, this part of our work focuses on "update-query" objects. On the one hand, the updates have a side-effect that usually affects the state of the object (hence all processes), but return no value. They correspond to transitions between abstract states in the transition system. On the other hand, the queries are read-only operations. They produce an output that depends on the state of the object. Consequently, the input alphabet of the transition system is separated into two classes of operations (updates and queries).

Definition 1 (Update-query abstract data type)**.** An update-query abstract data type (UQ-ADT) is a tuple $O = (U, Q_i, Q_o, S, s_0, T, G)$ such that:

- U is a countable set of *update* operations;
- Q_i and Q_o are countable sets called *input* and *output* alphabets; $Q = Q_i \times Q_o$ is the set of *query* operations. A query operation $(q_i, q_o) \in Q$ is denoted q_i/q_o (query q_i returns value q_o).
- S is a countable set of *states*;
- $s_0 \in S$ is the *initial state*;
- $T : S \times U \to S$ is the *transition function*;
- $G : S \times Q_i \to Q_o$ is the *output function*.

A sequential history is a sequence of operations. An infinite sequence of operations $(w_i)_{i \in \mathbb{N}} \in (U \cup Q)^\omega$ is recognized by O if there exists an infinite sequence of states $(s_i)_{i \geq 1} \in S^\omega$ (note that s_0 is the initial state) such that for all $i \in \mathbb{N}$, $T(s_i, w_i) = s_{i+1}$ if $w_i \in U$ or $s_i = s_{i+1}$ and $G(s_i, q_i) = q_o$ if $w_i = q_i/q_o \in Q$. The set of all infinite sequences recognized by O and their finite prefixes is denoted by $L(O)$. Said differently, $L(O)$ is the set of all the sequential histories allowed for O.

In the following, we use replicated sets as the key example. Three kinds of operations are possible: two update operation by element, namely insertion (I) and deletion (D) and a query operation read (R) that returns the values that belong to the set. Let *Val* be the support of the replicated set (it contains the values that can be inserted/deleted). At the beginning, the set is empty and when an element is inserted, it becomes present until it is deleted. More formally, it corresponds to the UQ-ADT given in Example 1.

Example 1 (Specification of the set)**.** Let *Val* be a countable set, called support. The set object $\mathcal{S}_{Val}$ is the UQ-ADT $(U, Q_i, Q_o, S, \emptyset, T, G)$ with:

- $U = \{\mathrm{I}(v), \mathrm{D}(v) : v \in Val\}$;
- $Q_i = \{\mathrm{R}\}$, and $Q_o = S = \mathcal{P}_{<\infty}(Val)$ contain all the finite subsets of *Val*;
- for all $s \in S$ and $v \in Val$, $G(s, \mathrm{R}) = s$,
  $T(s, \mathrm{I}(v)) = s \cup \{v\}$ and $T(s, \mathrm{D}(v)) = s \setminus \{v\}$.

The set U of updates is the set of all insertions and deletions of any value of *Val*. The set of queries Q_i contains a unique operation R, a read operation with no parameter. A read operation may return any value in Q_o, the set of all finite subsets of *Val*. The set S of the possible states is the same as the set of possible returned values Q_o as the read query returns the content of the set object. $I(v)$ (resp. $D(v)$) with $v \in Val$ denotes an insertion (resp. a deletion) operation of the value v into the set object. R/s denotes a read operation that returns the set s representing the content of the set.

During an execution, the participants invoke an object instance of an abstract data type using the associated operations (queries and updates). This execution produces a set of partially ordered events labelled by the operations of the abstract data type. This representation of a distributed history is generic enough to model a large number of distributed systems. For example, in the case of communicating sequential processes, an event a precedes an event b in the *program order* if they are executed by the same process in that sequential order. It is also possible to model more complex modern systems in which new threads are created and destroyed dynamically, or peer-to-peer systems where peers may join and leave.

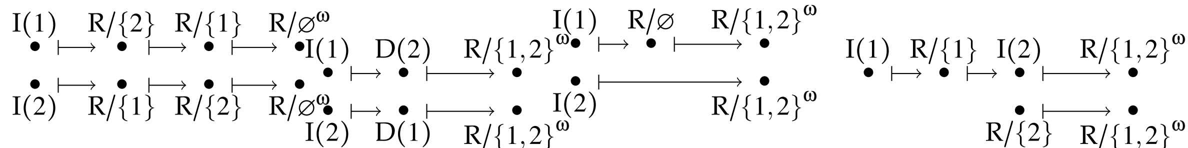
Definition 2 (Distributed History)**.** A distributed history is a tuple $H = (U, Q, E, \Lambda, \mapsto)$:

- U and Q are disjoint countable sets of *update* and *query* operations, and all queries $q \in Q$ are in the form $q = q_i/q_o$;
- E is a countable set of *events*;
- $\Lambda : E \to U \cup Q$ is a *labelling function*;
- $\mapsto \subset E \times E$ is a partial order called *program order*, such that for all $e \in E$, $\{e' \in E : e' \mapsto e\}$ is finite.

Let $H = (U, Q, E, \Lambda, \mapsto)$ be a history. The sets $U_H = \{e \in E : \Lambda(e) \in U\}$ and $Q_H = \{e \in E : \Lambda(e) \in Q\}$ denote its sets of update and query events respectively. We also define some projections on the histories. The first one allows to withdraw some events: for $F \subset E$, $H_F = (U, Q, F, \Lambda|_F, \mapsto \cap (F \times F))$ is the history that contains only the events of F. The second one allows to substitute the order relation: if $\to$ is a partial order that respects the definition of a program order ($\mapsto$), $H^{\to} = (U, Q, E, \Lambda, \to \cap (E \times E))$ is the history in which the events are ordered by $\to$. Note that the projections commute, which allows the notation $H_F^{\to}$.

Definition 3 (Linearizations)**.** Let $H = (U, Q, E, \Lambda, \mapsto)$ be a distributed history. A linearization of H corresponds to a sequential history that contains the same events as H in an order consistent with the program order. More precisely, it is a word $\Lambda(e_0) \ldots \Lambda(e_n) \ldots$ such that $\{e_0, \ldots, e_n, \ldots\} = E$ and for all i and j, if $i < j$, $e_j \not\mapsto e_i$. We denote by $\mathrm{lin}(H)$ the set of all linearizations of H.

Definition 4 (Consistency criterion)**.** A consistency criterion C characterizes which histories are allowed for a given data type. It is a function C that associates with any UQ-ADT O, a set of distributed histories $C(O)$. A shared object (instance of an UQ-ADT O) is C-consistent if all the histories it allows are in $C(O)$.



(a) EC but not SEC nor UC (b) SEC but not UC (c) SEC and UC but not SUC (d) SUC but not PC

Figure 6: Four histories for an instance of $\mathcal{S}_{\mathbb{N}}$ (cf. example 1), with different consistency criteria. The arrows represent the program order, and an event labeled ω is repeated an infinite number of times.

4.2 Eventual Consistency

In this section, we recall the definitions of eventual consistency [Vog08] and strong eventual consistency [SPB+11]. Fig. 6 illustrates these two consistency criteria on small examples. In the remaining of this article, we consider an UQ-ADT $O = (U, Q_i, Q_o, S, s_0, T, G)$ and a history $H = (U, Q, E, \Lambda, \mapsto)$.

Eventual consistency eventual consistency requires that, if all the participants stop updating, all the replicas eventually converge to the same state. In other word, H is eventually consistent if it contains an infinite number of updates (i.e. the participants never stop writing) or if there exists a state (the consistent state) compatible with all but a finite number of queries.

Definition 5 (Eventual consistency)**.** A history H is eventually consistent (EC) if U_H is infinite or there exists a state $s \in S$ such that the set of queries that return non consistent values while in the state s, $\{q_i/q_o \in Q_H : G(s, q_i) \neq q_o\}$, is finite.

All the histories presented in Fig. 6 are eventually consistent. The executions represent two processes sharing a set of integers. In Fig. 6a, the first process inserts value 1 and then reads twice the set and gets respectively $\{2\}$ and $\{1\}$; afterwards, it executes an infinity of read operations that return the empty set (ω in superscript denotes the operation is executed an infinity of times). In the meantime, the second process inserts a 2 then reads the set an infinity of times. It gets respectively $\{1\}$ and $\{2\}$ the two first times, and empty set an infinity of times. Both processes converge to the same state ($\emptyset$), so the history is eventually consistent. However, before converging, the processes can read anything a finite but unbounded number of times.

Strong eventual consistency strong eventual consistency requires that two replicas of the same object converge as soon as they have received the same updates. The problem with that definition is that the notions of replica and message reception are inherent to the implementation, and are hidden from the programmer that uses the object, so they should not be used in its specification. A visibility relation is introduced to model the notion of message delivery. This relation is not an order since it is not required to be transitive.

Definition 6 (Strong eventual consistency)**.** A history H is strong eventually consistent (SEC) if there exists an acyclic and reflexive relation $\xrightarrow{vis}$ (called *visibility* relation) that contains $\mapsto$ and such that:

- Eventual delivery: when an update is viewed by a replica, it is eventually viewed by all replicas, so there can be at most a finite number of operations that do not view it:
  $\forall u \in U_H, \{e \in E, u \not\xrightarrow{vis} e\}$ is finite;
- Growth: if an event has been viewed once by a process, it will remain visible forever:
  $\forall e, e', e'' \in E, (e \xrightarrow{vis} e' \wedge e' \mapsto e'') \Rightarrow (e \xrightarrow{vis} e'')$;
- Strong convergence: if two query operations view the same past of updates V, they can be

$$\begin{array}{ccccccccc} \mathrm{I}(1) & & \mathrm{I}(3) & & \mathrm{R}/\{1,3\} & & \mathrm{R}/\{1,2,3\} & & \mathrm{R}/\{1,2\}^\omega \\ \bullet_a & \longmapsto & \bullet_b & \longmapsto & \bullet_c & \longmapsto & \bullet_d & \longmapsto & \bullet_e \\ \bullet^f & \longmapsto & \bullet^g & \longmapsto & \bullet^h & \longmapsto & \bullet^i & \longmapsto & \bullet^j \\ \mathrm{I}(2) & & \mathrm{D}(3) & & \mathrm{R}/\{2\} & & \mathrm{R}/\{1,2\} & & \mathrm{R}/\{1,2,3\}^\omega \end{array}$$

$$w_1 = \mathrm{I}(1)\cdot\mathrm{I}(3)\cdot\mathrm{R}/\{1,3\}\cdot\mathrm{I}(2)\cdot\mathrm{R}/\{1,2,3\}\cdot\mathrm{D}(3)\cdot\mathrm{R}/\{1,2\}^\omega$$
$$w_2 = \mathrm{I}(2)\cdot\mathrm{D}(3)\cdot\mathrm{R}/\{2\}\cdot\mathrm{I}(1)\cdot\mathrm{R}/\{1,2\}\cdot\mathrm{I}(3)\cdot\mathrm{R}/\{1,2,3\}^\omega$$

Figure 7: PC but not EC

issued in the same state s: $\forall V \subset U_H, \exists s \in S, \forall q_i/q_o \in Q_H$,
$V = \{u \in U_H : u \xrightarrow{vis} q_i/q_o\} \Rightarrow G(s,q_i) = q_o$.

The history of Fig. 6a is not strong eventually consistent because the $\mathrm{I}(1)$ must be visible by all the queries of the first process (by reflexivity and growth), so there are only two possible sets of visible updates ($\{\mathrm{I}(1)\}$ and $\{\mathrm{I}(1),\mathrm{I}(2)\}$) for these events, but the queries are done in three different states ($\{1\}$, $\{2\}$ and $\emptyset$); consequently, at least two of these queries see the same set of updates and thus need to return the same value. Fig. 6c, on the contrary, is strong eventually consistent: the replicas that see $\{\mathrm{I}(1)\}$ are in state $\emptyset$ and those that see $\{\mathrm{I}(1),\mathrm{I}(2)\}$ are in state $\{1,2\}$.

4.3 Pipelined Convergence

A straightforward way to strengthen eventual consistency is to compose it with another consistency criterion that imposes restrictions on the values that can be returned by a read operation. Causality is often cited as a possible candidate to play this role [SJZ+98]. As causal consistency is well formalized only for memory, we will instead consider Pipelined Random Access Memory (PRAM) [LS88], a weaker consistency criterion. As the name suggests, PRAM was initially defined for memory. However, it can be easily extended to all UQ-ADTs. Let's call this new consistency criterion *pipelined consistency (PC)*. In a pipelined consistent computation, each process must have a consistent view of its local history with all the updates of the computation. More formally, it corresponds to Def. 7. Pipelined consistency is local to each process, as different processes can see concurrent updates in a different order.

Definition 7. A history H is *pipelined consistent* (PC) if, for all maximal chains (i.e. sets of totally ordered events) p of H, $\mathrm{lin}\left(H_{U_H \cup p}\right) \cap L(O) \neq \emptyset$.

Pipelined consistency can be implemented at a very low cost in wait-free systems. Indeed, it only requires FIFO reception. However, it does not imply convergence. For example, the history given in Figure 7 is pipelined consistent but not eventually consistent. In this history, two processes p_1 and p_2 share a set of integers. Process p_1 first inserts 1 and then 3 in the set and then reads the set forever. Meanwhile, process p_2 inserts 2, deletes 3 and reads the set forever. The words w_1 and w_2 are correct linearizations for both processes, with regard to Definition 7 so the history is pipelined consistent, but after stabilization, p_2 sees the element 3 whereas p_1 does not.

Proposition 1 (Implementation)**.** *Pipelined convergence, that imposes both pipelined consistency and eventual consistency, cannot be implemented in a wait-free system.*

Proof. We consider the same program as in Figure 7, and we suppose the shared set is pipelined convergent. By the same argument as developed in [AW94], it is not possible to prevent the processes from not seeing each other's first update at their first reads. Indeed, if p_1 did not receive any message from process p_2, it is impossible for p_1 to make the difference between the case where p_2 crashed before sending any message and the case where all its messages were delayed. To achieve availability, p_1 must compute the return value based solely on its local knowledge, so it returns $\{1,3\}$. Similarly, p_2 returns $\{2\}$. To circumvent this impossibility, it is necessary to make synchrony assumption on the system (e.g. bounds on transmission delays) or to assume the correctness of a majority of processes.

If the first read of p_1 returns $\{1,3\}$, as the set is pipelined consistent, there must exist a linearization for p_1 that contains all the updates, $R/\{1,3\}$ and an infinity of queries. As $2 \notin \{1,3\}$, the possible linearizations are defined by the ω-regular language $I(1)\cdot I(3)\cdot R/\{1,3\}^+\cdot I(2)\cdot R/\{1,2,3\}^\star\cdot D(3)\cdot R/\{1,2\}^\omega$, so any history must contain an infinity of events labelled $R/\{1,2\}^\omega$. Similarly, if p_2 starts by reading $\{2\}$, it will eventually read $\{1,2,3\}$ an infinity of times. This implies that pipelined convergence cannot be provided in wait-free systems. □

Consequently causal consistency, that is stronger than pipelined consistency, cannot be satisfied together with eventual consistency in a wait-free system.

5 Update Consistency

We now introduce our new consistency criteria: update consistency and strong update consistency. Section 5.1 provides their main definitions, while Section 5.2 discusses their expressive power by means of a case study.

5.1 Definitions

We give below the definitions of consistency and strong update consistency. Then, in Figure 6, we compare them to eventual consistency and strong eventual consistency on four small examples.

Update consistency eventual consistency and strong eventual consistency are not interested in defining the states that are reached during the histories (the same updates have to lead to the same state whatever is the state). They do not depend on the sequential specification of the object, so they give very little constraints on the histories. For example, an implementation that ignores all the updates is strong eventually consistent, as all the queries return the initial state. In update consistency, we impose the existence of a total order on the updates, that contains the program order and that leads to the consistent state according to the abstract data type. Another equivalent way to approach update consistency is that, if the number of updates is finite, it is possible to remove a finite number of queries such that the history is sequentially consistent.

Definition 8 (Update consistency)**.** A history H is update consistent (UC) if U_H is infinite or if there exists a finite set of queries $Q' \subset Q_H$ such that $\mathrm{lin}\left(H_{E\setminus Q'}\right) \cap L(O) \neq \emptyset$.

The history of Fig. 6c is update consistent because the sequence of operations I(1)I(2) is a possible explanation for the state $\{1,2\}$. The history of Fig. 6b is not update consistent because any linearization of the updates would position a deletion as the last event. Only three consistent states are actually possible: state $\emptyset$, e.g. for the linearization $I(1)\cdot I(2)\cdot D(1)\cdot D(2)$, state $\{1\}$ for the linearization $I(2)\cdot D(1)\cdot I(1)\cdot D(2)$ and state $\{2\}$ for the linearization $I(1)\cdot D(2)\cdot I(2)\cdot D(1)$. Update consistency is incomparable with strong eventual consistency.

Strong update consistency strong update consistency is a strengthening of both update consistency and strong eventual consistency. The relationship between update consistency and strong update consistency is analogous to the relation between eventual consistency and strong eventual consistency.

Definition 9 (Strong update consistency)**.** A history H is strong update consistent (SUC) if there exists (1) an acyclic and reflexive relation $\xrightarrow{vis}$ that contains $\mapsto$ and (2) a total order $\leq$ that contains $\xrightarrow{vis}$ such that:

- Eventual delivery:
  $\forall u \in U_H, \{e \in E, u \overset{vis}{\not\longrightarrow} e\}$ is finite;
- Growth:
  $\forall e, e', e'' \in E, \left(e \xrightarrow{vis} e' \wedge e' \mapsto e''\right) \Rightarrow \left(e \xrightarrow{vis} e''\right);$
- Strong sequential convergence: A query views an update if this update precedes it according to $\xrightarrow{vis}$. Each query is the result of the ordered execution, according to $\leq$, of the updates it views:
  $\forall q \in Q_H, \mathrm{lin}\left(H^{\leq}_{V(q)\cup\{q\}}\right) \cap L(O) \neq \varnothing$
  where $V(q) = \{u \in U_H : u \xrightarrow{vis} q\}$.

Fig. 6d shows an example of strong update consistent history: nothing prevents the second process from seeing the insertion of 2 before that of 1. Strong eventual consistency and update consistency does not imply strong update consistency: in the history of Fig. 6c, after executing event I(1), the only three possible update linearizations are I(1), I(1)·I(2) and I(2)·I(1) and none of them can lead to the state $\varnothing$ according to the sequential specification of a set object. So the history of Fig. 6c is not strong update consistent, while it is update consistent and strong eventually consistent.

Proposition 2 (Comparison of consistency criteria)**.** *If a history H is update consistent, then it is eventually consistent. If H is strong update consistent, then it is both strong eventually consistent and update consistent.*

Proof. Suppose H is update consistent. If H contains an infinite number of updates, then it is eventually consistent. Otherwise, there exists a finite set $Q' \subset Q_H$ and a word $w \in \mathrm{lin}\left(H_{E_H \setminus Q'}\right) \cap L(O)$. As the number of updates is finite, there is a finite prefix v of w that contains them all. $v \in L(O)$, so it labels a path between s_0 and a state s in the UQ-ADT. All the queries that are in w but not in v return the same state s, and the number of queries in Q' and v is finite. Hence, H is eventually consistent.

Suppose H is strong update consistent with a finite number of updates. $Q' = \bigcup_{u \in U_H} \{q \in Q_H, q \leq u\}$ is finite, and $\mathrm{lin}(E_H \setminus Q')$ contains only one word that is also contained into $L(O)$. Obviously, H is update consistent

Now, suppose H is strong update consistent. Strong update consistency respects both eventual delivery and growth properties. Let $V \subset U_H$. As the relation $\leq$ is a total order, there is a unique word w in $\mathrm{lin}\left(H^{\leq}_V\right) \cap L(O)$. Let us denote s the state obtained after the execution of w. For all $q \in Q_H$ such that $V = \{u \in U_H : u \xrightarrow{vis} q\}$, $\mathrm{lin}\left(H^{\leq}_{E_q \cup \{q\}}\right) \cap L(O) = \{w \cdot \Lambda(q)\}$, so $q = q_i/q_o$ with $G(s, q_i) = q_o$. Consequently, H is strong eventually consistent. □

5.2 Expressiveness of Update Consistency: a Case Study

The set is one of the most studied eventually consistent data structures. Different types of sets have been proposed as extensions to CRDTs to implement eventually consistent sets even though the insert and delete operations do not commute. The simplest set is the Grow-Only Set (G-Set) [SPB+11], in which it is only possible to insert elements. As the insertion of two elements commute, G-Set is a CRDT. Using two G-Set, a white list for inserted elements and a black list for the deleted ones, it is possible to build a Two-Phases Set (2P-Set, a.k.a. U-Set, for Unique Set) [WB86], in which it is possible to insert and remove elements, but never insert again an element that has already been deleted. Other implementations such as C-Set [AMSM+11] and PN-Set, add counters on the elements to determine if they should be present or not. The Observe-Remove Set (OR-Set) [SPB+11, MSS14] is the best documented alg orithm for the set. It is very close to the 2P-Set in its principles, but each insertion is timestamped with a unique identifier, and the deletion only black-lists the identifiers that it observes. It guaranties that, if an insertion and a deletion of the same element are concurrent, the insertion will win and the element will be added to the set. Finally, the last-writer-wins element set (LWW-element-Set) [SPB+11] attaches a timestamp to each element to decide which operation should win in case of conflict. All these sets, and the eventually consistent objects in general, have a different behavior when they are used in distributed programs.

The above mentioned implementations are eventually consistent. However, as eventual consistency does not impose a semantic link between updates and queries, it is hazardous to say anything on the conformance to the specification of the object. Burckhardt *et al.* [BGYZ14] propose to specify the semantics of a query by a function on its concurrent history, called *visibility*, that corresponds to the visibility relation in strong eventual consistency, and a linearization of this history, called *arbitration*. In comparison, sequential specifications are restricted to the arbitration relation. It implies that fewer update consistent objects than eventually consistent objects can be specified. Although the variety of objects with a distributed specification seems to be a chance that compensates the lower level of abstraction it allows, an important bias must be taken into account: from the point of view of the user, the visibility of an operation is not an *a priori* property of the system, but an *a posteriori* way to explain what happened. If one only focuses on the final state, an update consistent object is appropriate to be used instead of an eventually consistent object, since the final state is the same as if no operations were concurrent.

By adding further constraints on the histories, concurrent specifications strengthen the consistency criteria. Even if strong update consistency is stronger than strong eventual consistency, we cannot say in general that a strong update consistent object can always be used instead of its strong eventually consistent counterpart. We claim that this is true in practice for *reasonable* objects, and we prove this in the case of the Insert-wins set (the concurrent specification of the OR-set). The arbitration relation is not used for the OR-set, and the visibility relation has already been defined for strong eventual consistency. The concurrent specification only adds one more constraint on this relation: an element is present in the set if and only if it was inserted and is not yet deleted.

Definition 10 (Strong eventual consistency for the Insert-wins set)**.** A history H is strong eventually consistent for the Insert-wins set on a support Val if it is strong eventually consistent for the set $\mathcal{S}_{Val}$ and the visibility relation $\xrightarrow{vis}$ verifies the following additional property. For all $x \in Val$ and $q \in Q_H$, with $\Lambda(q) = R/s$, $x \in s \Leftrightarrow \left(\exists u \in vis(q, \mathrm{I}(x)), \forall u' \in vis(q, \mathrm{D}(x)), u \not\xrightarrow{vis} u'\right)$, where for all $o \in U$, $vis(q,o) = \{u \in U_H : u \xrightarrow{vis} q \wedge \Lambda(u) = o\}$.

The OR-Set implementation of a set is not update consistent. The history on Fig. 6b is not update consistent, as the last operation must be a deletion. However, if the updates made by a process are not viewed by the other process before it makes its own updates, the insertions will win and the OR-set will converge to $\{1,2\}$. On the contrary, a strong update consistent implementation of a set can always be used instead of an Insert-wins set, as it only forbids more histories.

Proposition 3 (Comparison with Insert-wins set)**.** *Let $H = (U,Q,E,\Lambda,\mapsto)$ be a history that is strong update consistent for $\mathcal{S}_{Val}$. Then H is strong eventually consistent for the Insert-wins set.*

Proof. Suppose H is strong update consistent for $\mathcal{S}_{Val}$. We define the new relation $\xrightarrow{IW}$ such that for all $e,e' \in E$, $e \xrightarrow{IW} e'$ if one of the following conditions holds:

- $e \xrightarrow{vis} e'$;
- e and e' are two updates on the same element and $e \leq e'$;

- e' is a query, and there is an update e'' such that $e \xrightarrow{IW} e''$ and $e'' \xrightarrow{IW} e'$.

The relation $\xrightarrow{IW}$ is acyclic because it is included in $\leq$, its growth and eventual delivery properties are ensured by the fact that it contains $\xrightarrow{vis}$. Moreover, no two updates for the same element are concurrent according to $\xrightarrow{IW}$ and the last updates are also the last for the $\leq$ relation, consequently H is strong eventually consistent for the Insert-wins set. $\square$

This result implies that an OR-set can always be replaced by an update consistent set, because the guaranties it ensures are weaker than those of the update consistent set. It does not mean that the OR-set is worthless. It can be seen as a cache consistent set [Goo91] that, in some cases may have a better space complexity than update consistency.

6 Generic Construction of Strong Update Consistent Objects

In this section, we give a generic construction of strong update consistent objects in crash-prone asynchronous message-passing systems. This construction is not the most efficient ever as it is intended to work for any UQ-ADT object in order to prove the universality of update consistency. For a specific object an ad hoc implementation on a specific system may be more suitable.

6.1 System Model

We consider a message-passing system composed of finite set of sequential *processes* that may fail by halting. A faulty process simply stops operating. A process that does not crash during an execution is correct. We make no assumption on the number of failures that can occur during an execution. Processes communicate by exchanging *messages* using a communication network complete and reliable. A message sent by a correct process to another correct process is eventually received. The system is asynchronous; there is no bound on the relative speed of processes nor on the message transfer delays. In such a situation a process cannot wait for the participation of any a priori known number of processes as they can fail. Consequently, when an operation on a replicated object is invoked locally at some process, it needs to be completed based solely on the local knowledge of the process. We call this kind of systems wait-free asynchronous message-passing system.

We model executions as histories made up of the sequences of events generated by the different processes. As we focus on shared objects and their implementation, only two kinds of actions are considered: the operations on shared objects, that are seen as events in the distributed history, and message receptions.

6.2 A universal implementation

Now, we prove that strong update consistency is universal, in the sense that every UQ-ADT has a strong update consistent implementation in a wait-free asynchronous system. Algorithm 3 presents an implementation of a generic UQ-ADT. The principle is to build a total order on the updates on which all the participants agree, and then to rewrite the history *a posteriori* so that every replica of the object eventually reaches the state corresponding to the common sequential history. Any strategy to build the total order on the updates would work. In Algorithm 3, this order is built from a Lamport's clock [Lam78] that contains the happened-before precedence relation. Process order is hence respected. A logical Lamport's clock is a pre-total order as some events may be associated with the same logical time. In order to have a total order, the events are timestamped with a pair composed of the logical time and the id of the process that produced it (process ids are assumed unique and totally ordered). The algorithm actions performed by a process p_i are atomic and totally ordered by an order $\mapsto_i$. The union of these orders for all processes is the program order $\mapsto$.

Algorithm 3 a generic UQ-ADT (code for p_i)

```
object (U, Q_i, Q_o, S, s_0, T, G)
  var clock_i ∈ ℕ ← 0;
  var update_i ⊂ (ℕ × ℕ × U) ← ∅;
  fun update (u ∈ U)
    clock_i ← clock_i + 1;
    broadcast message (clock_i, i, u);
  end
  on receive message (cl ∈ ℕ, j ∈ ℕ, u ∈ Q)
    clock_i ← max(clock_i, cl);
    update_i ← update_i ∪ {(cl, j, u)};
  end
  fun query (q ∈ Q_i) ∈ Q_o
    clock_i ← clock_i + 1;
    var state_i ∈ S ← s_0;
    for (cl, j, u) ∈ update_i sorted on (cl, j) do
      state_i ← T(state_i, u);
    end
    return G(state_i, q);
  end
end
```

At the application level, a history is composed of update and query operations. In order to allow only strong update consistent histories, Algorithm 3 proposes a procedure $update()$ and a function $query()$. A history H is allowed by the algorithm if $update(u)$ is called each time a process performs an update u, and $query(q_i)$ is called and returns q_o when the event q_i/q_o appears in the history. The code of Algorithm 3 is given for process p_i. Each process p_i manages its view $clock_i$ of the logical clock and a list $updates_i$ of all timestamped update events process p_i is aware of. The list $updates_i$ contains triplets (cl, j, u) where u is an update event and (cl, j) the associated timestamp. This list is sorted according to the timestamps of the updates: $(cl, j) < (cl', j')$ if $(cl < cl')$ or $(cl = cl'$ and $j < j')$.

The algorithm timestamps all events (updates and queries). When an update is issued locally, process p_i informs all the other processes by reliably broadcasting a message to all other processes (including itself). Hence, all processes will eventually be aware of all updates. When a $message(cl, j, u)$ is received, p_i updates its clock and inserts the event to the list $updates_i$. When a query is issued, the function $query()$ replays locally the whole list of update events p_i is aware of starting from the initial state then it executes the query on the state it obtains.

Whenever an operation is issued, its is completed without waiting for any other process. This corresponds to wait-free executions in shared memory distributed systems and implies fault-tolerance.

Proposition 4 (Strong update consistency)**.** *All histories allowed by Algorithm 3 are strong update consistent.*

Proof. Let $H = (U, Q, E, \Lambda, \mapsto)$ be a distributed history allowed by Algorithm 3. Let $e, e' \in E_H$ be two operations invoked by processes p_i and $p_{i'}$, on the states (`update`,`clock`) and (`update'`,`clock'`), respectively. We pose:

- $e \xrightarrow{vis} e'$ if $e \in U_H$ and $p_{i'}$ received the message sent during the execution of e before it starts executing e', or $e \in Q_H$ and $e \mapsto e'$. As the messages are received instantaneously by the sender, $\xrightarrow{vis}$ contains $\mapsto$. It is growing because the set of messages received by a process is growing with time.
- $e \leq e'$ if $c < c'$ or $c = c'$ and $i < i'$. This lexical order is total because two operations on the same process have a different clock. Moreover it contains $\xrightarrow{vis}$ because when $p_{i'}$ received the

Algorithm 4 the shared memory (code for p_i)

```
1  object UC_mem(X, V, v₀)
2      var clockᵢ ∈ ℕ ← 0;
3      var memᵢ ∈ mem(X, (ℕ² × V), (0, 0, v₀));
4      fun write (x ∈ X, v ∈ V)
5          clockᵢ ← clockᵢ + 1;
6          broadcast msg (clockᵢ, i, x, v);
7      end
8      on receive msg (cl ∈ ℕ, j ∈ ℕ, x ∈ X, v ∈ V)
9          clockᵢ ← max(clockᵢ, cl);
10         var (cl′, j′, v′) ∈ ℕ² × V ← memᵢ.read(x);
11         if (cl′, j′) < (cl, j) then
12             memᵢ.write(x, (cl, j, v))
13         end
14     end
15     fun read (x ∈ X) ∈ V
16         var (cl, j, v) ∈ ℕ² × V ← memᵢ.read(x);
17         return v;
18     end
19 end
```

message sent by e, it executed line 9 and when it executed e', it executed line 5, so $c' \geq c+1$. Moreover, the history of e contains at most $c \times n + i$ events, where n is the number of processes, so it is finite.

Let $q \in Q_H$ and $E_q = \{u \in U_H : u \xrightarrow{vis} q\}$. Lines 15 to 18 build an explicit sequential execution, that is in $\mathrm{lin}\left(H^{\leq}_{E_q \cup \{q\}}\right)$ by definition of $\leq$ and in $L(O)$ by definition of O. $\square$

6.3 Complexity

Algorithm 3 is very efficient in terms of network communication. A unique message is broadcast for each update and each message only contains the information to identify the update and a timestamp composed of two integer values, that only grow logarithmically with the number of processes and the number of operations. Moreover, this algorithm is wait-free and its execution does not depend on the latency of the network.

This algorithm re-executes all past updates each time a new query is issued. In an effective implementation, a process can keep intermediate states. These intermediate states are re-computed only if very late message arrive. The algorithm does not look space efficient also as the whole history must be kept in order to rebuild a sequential history. Because data space is cheap and fast nowadays, compared to bandwidth, many applications can afford this complexity and would keep this information anyway. For example, banks keep track of all the operations made on an account for years for legal reasons. In databases systems, it is usual to record all the events in log files. Moreover, asynchrony is used as a convenient abstraction for systems in which transmission delays are actually bounded, but the bound is too large to be used in practice. This means that after some time old messages can be garbage collected.

The proposed algorithm is a theoretical work whose goal is to prove that any update-query object has a strong update consistent implementation. This genericity prevents an effective implementation that may take benefit from the nature and the specificity of the actual object. The best example of this are pure CRDTs like the counter and the grow-only set. If all the update operations commute in the sequential specification, all linearizations would lead to the same state so a naive implementation, that applies the updates on a replica as soon as the notification is received, achieves update consistency. In [KBL93], Karsenty and Beaudouin-Lafon propose an algorithm to implement objects such that each update operation u contains an *undo* u^{-1} such that for all s,

$T(T(s,u),u^{-1}) = s$. This algorithm is very close to ours as it builds the convergent state from a linearization of the updates stored by each replica. They use the undo operations to position newly know n updates at their correct place, which saves computation time. As it is a very frequent example in distributed systems, we now focus on the shared memory object.

Algorithm 4 shows an update consistent implementation of the shared memory object. A shared memory offers a set X of registers that contain values taken from a set V. The query operation read(x), where $x \in X$, returns the last value $v \in V$ written by the update operation write(x,v), or the initial value $v_0 \in V$ if x was never written. Algorithm 4 orders the updates exactly like Algorithm 3. As the old values can never be read again, it is not necessary to store them forever, so the algorithm only keeps in memory the last known value of each register and its timestamp in a local memory mem_i, implemented with an associative array. When a process receives a notification for a write, it updates its local state if the value is newer that the current one, and the read operations just return the current value. This implementation only needs constant computation time for both the reads and the writes, and the complexity in memory only grows logarithmically with time and the number of participants.

7 Conclusion

This work was motivated by the increasing popularity of geographically distributed systems. We have presented two contributions that make it possible to formally define and reason about consistency conditions in large-scale systems. The first contribution defines a mixed approach in which the operations invoked by nearby processes obey stronger consistency requirements than operations invoked by remote ones. The second consists of a new consistency criterion, update consistency, that is stronger than eventual consistency and weaker than sequential consistency. Update consistency formalizes the intuitive notions of sequential specification for an abstract data type and distributed history.

References

[ABD95] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *J. ACM*, 42(1):124–142, 1995.

[AF92] Hagit Attiya and Roy Friedman. A correctness condition for high-performance multiprocessors (extended abstract). In *Proceedings of the Twenty-fourth Annual ACM Symposium on Theory of Computing*, STOC '92, pages 679–690, New York, NY, USA, 1992. ACM.

[AMSM+11] Khaled Aslan, Pascal Molli, Hala Skaf-Molli, Stéphane Weiss, et al. C-set: a commutative replicated data type for semantic stores. In *RED: Fourth International Workshop on REsource Discovery*, 2011.

[ANB+95] Mustaque Ahamad, Gil Neiger, James E Burns, Prince Kohli, and Phillip W Hutto. Causal memory: Definitions, implementation, and programming. *Distributed Computing*, 9(1):37–49, 1995.

[AW94] Hagit Attiya and Jennifer L. Welch. Sequential consistency versus linearizability. *ACM Trans. Comput. Syst.*, 12(2):91–122, May 1994.

[AW04] Hagit Attiya and Jennifer Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. John Wiley & Sons, 2004.

[BGYZ14] Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. Replicated data types: specification, verification, optimality. In *Proceedings of the 41st* *annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 271–284. ACM, 2014.

[BJ87] Kenneth P. Birman and Thomas A. Joseph. Reliable communication in the presence of failures. *ACM Trans. Comput. Syst.*, 5(1):47–76, January 1987.

[Bre00] Eric A. Brewer. Towards robust distributed systems (abstract). In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '00, pages 7–, New York, NY, USA, 2000. ACM.

[BSS91] Kenneth Birman, André Schiper, and Pat Stephenson. Lightweight causal and atomic group multicast. *ACM Trans. Comput. Syst.*, 9(3):272–314, August 1991.

[BZP+12] Annette Bieniusa, Marek Zawirski, Nuno Preguiça, Marc Shapiro, Carlos Baquero, Valter Balegas, and Sérgio Duarte. An optimized conflict-free replicated set. *arXiv preprint arXiv:1210.3368*, 2012.

[DHJ+07] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 205–220. ACM, 2007.

[FTR15] R. Friedman, F. Taïani, and M. Raynal. Fisheye consistency: keeping data in synchr in a georeplicated world. In *Proc. Third Int'l Conference on Networked Systems (NETYS'15)*, pages 237–251. Springer LNCS 9476, 2015.

[Goo91] James R Goodman. *Cache consistency and sequential consistency*. University of Wisconsin-Madison, Computer Sciences Department, 1991.

[Her91] Maurice Herlihy. Wait-free synchronization. In *ACM Transactions on Programming Languages and Systems*, pages 124–149, 1991.

[HS08] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[KBL93] Alain Karsenty and Michel Beaudouin-Lafon. An algorithm for distributed groupware applications. In *ICDCS*, 1993.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[LM10] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010.

[LS88] Richard J Lipton and Jonathan S Sandberg. *PRAM: A scalable shared memory*. Princeton University, Department of Computer Science, 1988.

[Lyn96] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[LZM00] Du Li, Limin Zhou, and Richard R Muntz. A new paradigm of user intention preservation in realtime collaborative editing systems. In *International Conference on Parallel And Distributed Systems*, pages 401–408. IEEE, 2000.

[Mea55] George H Mealy. A method for synthesizing sequential circuits. *Bell System Technical Journal*, 34(5):1045–1079, 1955.

[MSS14] Madhavan Mukund, Gautham Shenoy, and SP Suresh. Optimized or-sets without ordering constraints. In *Distributed Computing and Networking*, pages 227–241. Springer, 2014.

[MZR95] Masaaki Mizuno, J.Z. Zhou, and Michel Raynal. Sequential consistency in distributed systems : theory and implementation. Technical Report RR-2437, INRIA, 1995.

[PMJ15] Matthieu Perrin, Achour Mostéfaoui, and Claude Jard. Update consistency for wait-free concurrent objects. In *2015 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2015, Hyderabad, India, May 25-29, 2015*, pages 219–228, 2015.

[PPJM14] Matthieu Perrin, Matoula Petrolia, Claude Jard, and Achour Mostéfaoui. Consistent shared data types: Beyond memory. Technical report, LINA, Université de Nantes, 2014.

[Ray12] Michel Raynal. *Concurrent Programming: Algorithms, Principles, and Foundations*. Springer Publishing Company, Incorporated, 2012.

[Ray13] Michel Raynal. *Distributed Algorithms for Message-Passing Systems*. Springer Publishing Company, Incorporated, 2013.

[RST91] Michel Raynal, André Schiper, and Sam Toueg. The causal ordering abstraction and a simple way to implement it. *Inf. Process. Lett.*, 39(6):343–350, October 1991.

[SJZ+98] Chengzheng Sun, Xiaohua Jia, Yanchun Zhang, Yun Yang, and David Chen. Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 5(1):63–108, 1998.

[SPAL11] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. Transactional storage for geo-replicated systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 385–400, New York, NY, USA, 2011. ACM.

[SPB+11] Marc Shapiro, Nuno Preguiça, Carlos Baquero, Marek Zawirski, et al. A comprehensive study of convergent and commutative replicated data types. Technical report, INRIA, 2011.

[SPBZ11] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *Stabilization, Safety, and Security of Distributed Systems*, pages 386–400. Springer, 2011.

[Vog08] Werner Vogels. Eventually consistent. *Queue*, 6(6):14–19, 2008.

[WB86] Gene TJ Wuu and Arthur J Bernstein. Efficient solutions to the replicated log and dictionary problems. *Operating systems review*, 20(1):57–66, 1986.

[XSK+14] Chao Xie, Chunzhi Su, Manos Kapritsos, Yang Wang, Navid Yaghmazadeh, Lorenzo Alvisi, and Prince Mahajan. Salt: Combining acid and base in a distributed database. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 495–509, Broomfield, CO, October 2014. USENIX Association.