Fisheye Consistency: Keeping Data in Synch in a Georeplicated World

R Friedman, M Raynal, François Taïani

Fisheye Consistency: Keeping Data in Synch in a Georeplicated World

R. Friedman[1], M. Raynal[2,3], and F. Taïani[3]

[1] The Technion Haifa, Israel
[2] Institut Universitaire de France
[3] Université de Rennes 1, IRISA, Campus de Beaulieu, 35042 Rennes Cedex, France

Abstract. Over the last thirty years, numerous consistency conditions for replicated data have been proposed and implemented. Popular examples include linearizability (or atomicity), sequential consistency, causal consistency, and eventual consistency. These 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. To address this lack, as a first contribution, this paper introduces the 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. The second contribution exploits this graph to provide a generic approach to the hybridization of data consistency conditions within the same system. We illustrate this approach on sequential consistency and causal consistency, and present a model in which all data operations are causally consistent, while operations by neighboring processes in the proximity graph are sequentially consistent. The third contribution of the paper is the design and the proof of a distributed algorithm based on this proximity graph, which combines sequential consistency and causal consistency (the resulting condition is called *fisheye consistency*). In doing so the paper provides a generic provably correct solution of direct relevance to modern georeplicated systems.
Keywords: Asynchronous message-passing systems, Broadcast, Causal consistency, Data replication, Georeplication, Linearizability, Sequential consistency.

1 Introduction

As distributed computer systems continue to grow in size, they make it increasingly difficult to provide strong consistency guarantees (e.g., linearizability [20]), prompting the rise of weaker guarantees (e.g., causal consistency [2] or eventual consistency [39]). These weaker consistency conditions strike a compromise between consistency, performance, and availability [5,7,10,13,40]. They try in general to minimize the violations of strong consistency, as these create anomalies for programmers and users, and emphasize the low probability of such violations in their real deployments [15].

Recent related works For brevity, we cannot name all the many weak consistency conditions that have been proposed in the past. We focus instead on the most recent

works in this area. One of the main hurdles in building systems and applications based on weak consistency models is how to generate an eventually consistent and meaningful image of the shared memory or storage [39]. In particular, a paramount sticking point is how to handle conflicting concurrent write (or update) operations and merge their result in a way that suits the target application. To that end, various conditions that enables custom conflict resolution and a host of corresponding data-types have been proposed and implemented [3,4,9,14,26,30,36,35].

Another form of hybrid consistency conditions can be found in the seminal works on *release consistency* [18,21] and *hybrid consistency* [6,16], which distinguish between strong and weak operations such that strong operations enjoy stronger consistency guarantees than weak operations. This line of work has given rise to a number of contributions in the context of large scale and geo-replicated data centers [38,40].

Motivation and problem statement 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 (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 [22], for instance, the application can specify for each operation a consistency guarantee that is dependent on the location of replicas. More precisely, the constraints LOCAL_QUORUM requires a quorum of replicas in the local data center, while EACH_QUORUM requires a quorum in each data center. Yet, although these constraints take distance into account, they do not provide the programmer with a precise image of the consistency they deliver.

The need to consider "distance" when defining 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 this paper (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 (a weaker one and a stronger one), taken from a set of totally ordered consistency conditions (e.g. linearizability, sequential consistency, causal consistency, and PRAM-consistency [25]).

The philosophy we advocate is related to that of Parallel Snapshot Isolation (PSI) proposed in [37]. 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.

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 distinguishes 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 [37] 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.

Roadmap The next section introduces the system model and the classical sequential consistency (SC) [24] and causal consistency (CC) [2]. Then, Section 3 defines the notion of *proximity graph* and the associated fisheye consistency condition, which considers SC as its strong condition and CC as its weak condition. Section 4 presents a broadcast abstraction, and Section 5 proposes an algorithm based on this broadcast abstraction that implements the fisheye consistency condition that combines SC and CC. These algorithms are generic, where the genericity parameter is the proximity graph. Interestingly, their two extreme instantiations provide natural implementations of SC and CC. Section 6 concludes.

2 System Model and Basic Consistency Conditions

The system consists of n processes denoted $\Pi = \{p_1, ..., p_n\}$. 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.

Processes communicate by passing messages through bi-directional channels. Channels are reliable (no loss, duplication, creation, or corruption), and asynchronous (transit times are arbitrary but finite, and remain unknown to the processes).

2.1 Basic notions and definitions underpinning consistency conditions

This section is a short reminder of the fundamental notions typically used to define the consistency guarantees of distributed objects [8,19,27,31].

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 [19,31]. 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, we restrict the discussion in this paper 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 Sequential consistency

Intuitively, an execution is sequentially consistent if it could have been produced by executing (with the help of a scheduler) the processes on a monoprocessor. Formally, a history $\widehat{H}$ is *sequentially consistent* (SC) if there exists a history $\widehat{S}$ such that:

- $\widehat{S}$ is sequential,
- $\widehat{S}$ is legal (the specification of each object is respected),

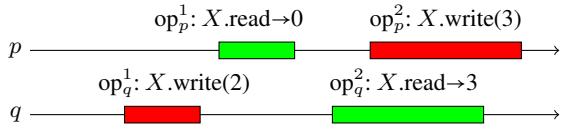

Fig. 1. A sequentially consistent execution

– $\widehat{H}$ and $\widehat{S}$ are equivalent (no process can distinguish $\widehat{H}$—what occurred—and $\widehat{S}$—what we would like to see, to be able to reason about).

One can notice that SC does not demand that the sequence $\widehat{S}$ respects the real-time occurrence order on the operations. This is the fundamental difference between linearizability and SC.

Figure 1 shows an history $\widehat{H}$ that is sequentially consistent. Let us observe that, although op_q^1 occurs before op_p^1 in physical time, op_p^1 does not see the effect of the write operation op_q^1, and still returns 0. A legal sequential history $\widehat{S}$, equivalent to $\widehat{H}$, can be easily built, namely, $X.\text{read} \rightarrow 0,\ X.\text{write}(2),\ X.\text{write}(3),\ X.\text{read} \rightarrow 3$.

2.3 Causal consistency

In a sequentially consistent execution, all processes perceive all operations in the same order, which is captured by the existence of a sequential and legal history $\widehat{S}$. Causal consistency [2] relaxes this constraint for read-write registers, and allows different processes to perceive different orders of operations, as long as causality is preserved.

Formally, a history $\widehat{H}$ in which processes interact through concurrent read/write registers is causally consistent (CC) if:

– There is a causal order $\rightsquigarrow_H$ on the operations of $\widehat{H}$, i.e., a partial order that links each read to at most one latest write (or otherwise to an initial value $\perp$), so that the value returned by the read is the one written by this latest write and $\rightsquigarrow_H$ respects the process order of all processes.
– For each process p_i, there is a sequential and legal history $\widehat{S}_i$ that
  • is equivalent to $\widehat{H}|(p_i + W)$, where $\widehat{H}|(p_i + W)$ is the sub-history of $\widehat{H}$ that contains all operations of p_i, plus the writes of all the other processes,
  • respects $\rightsquigarrow_H$ (i.e., $\rightsquigarrow_H \subseteq \rightarrow_{S_i}$).

Intuitively, this definition means that all processes see causally related writes in the same order, but can see writes that are not causally related in different orders.

An example of causally consistent execution is given in Figure 2. The processes r and s observe the write operations on X by p (op_p^1) and q (op_q^1) in two different orders. This is acceptable in a causally consistent history because op_p^1 and op_q^1 are not causally related. This would not be acceptable in a sequentially consistent history, where the same total order on operations must be observed by all the processes.

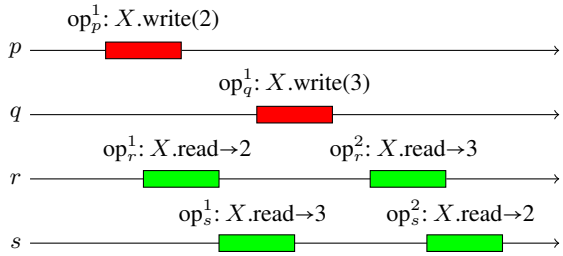

Fig. 2. An execution that is causally consistent (but not sequentially consistent)

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**

Fig. 3. *new-york* does not need to be closely synchronized with *paris* and *berlin*, calling for a hybrid form of consistency

3 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.

3.1 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 $\mathcal{G}$. $\mathcal{G}$ captures the level of consistency imposed on processes: processes connected in $\mathcal{G}$ must respect a stronger data consistency than unconnected processes.

Example To illustrate the semantic of $\mathcal{G}$, we extend the original scenario that Ahamad, Niger *et al* use to motivate causal consistency in [2]. Consider the three processes of Figure 3, *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.

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

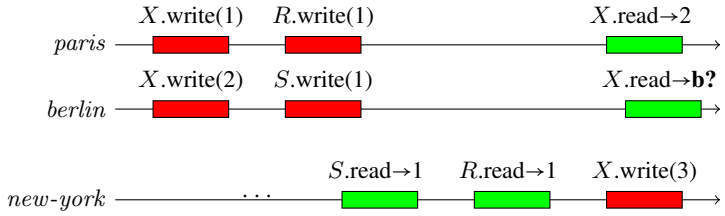

Fig. 4. Executing the program of Figure 3.

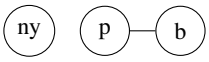

Fig. 5. Capturing the synchronization needs of Fig. 3 with a proximity graph $\mathcal{G}$

processes (because $new\text{-}york$ 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 4, 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 4, $berlin$ can only read 2 (the value it wrote) or 3 (written by $new\text{-}york$), but not 1. Sequential consistency is however too strong here: because the write operation of $new\text{-}york$ 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 [2]: $new\text{-}york$ is here free of concurrent write w.r.t. $paris$ and $berlin$, making causal consistency equivalent to sequential consistency for $new\text{-}york$. $paris$ 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 $new\text{-}york$ 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 $new\text{-}york$ 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 5: $paris$ and $berlin$ are connected, meaning the operations they execute should be perceived as strongly consistent w.r.t. one another ; $new\text{-}york$ is neither connected to $paris$ nor $berlin$, meaning a weaker consistency is allowed between the operations executed at $new\text{-}york$ and those of $paris$ and $berlin$.

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

When applied to the scenario of Figure 4, fisheye consistency combines two consistency conditions (a weak and a stronger 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 $\rightsquigarrow_H$ induced by $\widehat{H}$ (as in causal consistency); and
- $\rightsquigarrow_H$ 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

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

  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 4 with the proximity graph proposed in Figure 5 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 $new\text{-}york$ 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 $new\text{-}york$.

Given a system of n processes, let $\varnothing_\Pi = (\Pi, \varnothing)$ denote the graph with no edges, and K_Π denote the complete graph $(\Pi, \Pi \times \Pi)$. It is easy to see that CC is $\varnothing_\Pi$-fisheye (SC,CC)-consistency. Similarly SC is K_Π-fisheye (SC,CC)-consistency.

A larger example Figure 6 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 and then 3 for X, s must do the same, and only 3 can be returned by **x?**. For the same reason, only 3 can be returned by **y?**, as shown in the first line of Table 1.

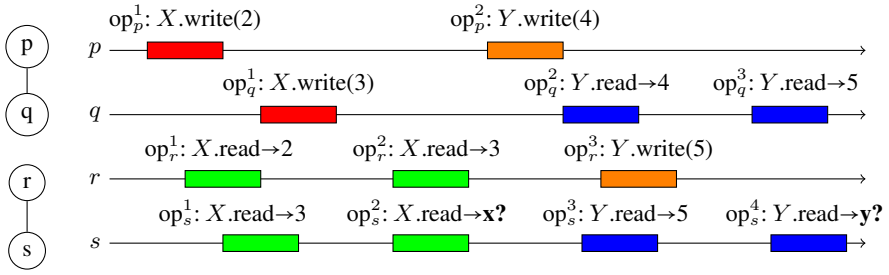

Fig. 6. Illustrating $\mathcal{G}$-fisheye (SC,CC)-consistency

Table 1. Possible executions for the history of Figure 6

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

In a causally consistent history, however, both pairs of writes ($\{\text{op}_p^1, \text{op}_q^1\}$ and $\{\text{op}_p^2, \text{op}_r^3\}$) 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 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).

4 Construction of an Underlying (SC,CC)-Broadcast Operation

Our implementation of $\mathcal{G}$-fisheye (SC,CC)-consistency uses a broadcast operation with hybrid ordering guarantees. We present here this hybrid broadcast, before moving on to the actual implementation of of $\mathcal{G}$-fisheye (SC,CC)-consistency in Section 5.

4.1 $\mathcal{G}$-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 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 TOCO_broadcast() and 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 $\rightsquigarrow_M$, is defined as follows [11,34]. Let $m_1, m_2 \in M$; $m_1 \rightsquigarrow_M m_2$, 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 $(m_1 \rightsquigarrow_M m) \wedge (m \rightsquigarrow_M m_2)$.

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 $m_1 \rightsquigarrow_M m_2$, 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 down to total order delivery.

4.2 $\mathcal{G}$-fisheye (SC,CC)-broadcast: algorithm

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 clocks 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 of messages received but not yet toco-delivered by p_i.

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 TO-COBC$(m, \langle causal_i[\cdot], total_i[i], i\rangle)$ 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

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** $total_i[i] \leq s\_tot_j^m$ **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** $C \neq \varnothing$ **where**
21: $C \equiv \Big\{\big\langle m, \langle s\_caus_j^m[\cdot], s\_tot_j^m, j\rangle\big\rangle \in pending_i \;\Big|\; s\_caus_j^m[\cdot] \leq causal_i[\cdot]\Big\}$
22: **wait until** $T_1 \neq \varnothing$ **where**
23: $T_1 \equiv \Big\{\big\langle m, \langle s\_caus_j^m[\cdot], s\_tot_j^m, j\rangle\big\rangle \in C \;\Big|\; \forall p_k \in N_{\mathcal{G}}(p_j) : \langle total_i[k], k\rangle > \langle s\_tot_j^m, j\rangle\Big\}$
24: **wait until** $T_2 \neq \varnothing$ **where**
25: $T_2 \equiv \left\{\big\langle m, \langle s\_caus_j^m[\cdot], s\_tot_j^m, j\rangle\big\rangle \in T_1 \;\middle|\; \begin{array}{l} \forall p_k \in N_{\mathcal{G}}(p_j), \\ \forall \big\langle m_k, \langle s\_caus_k^{m_k}[\cdot], s\_tot_k^{m_k}, k\rangle\big\rangle \\ \qquad \in pending_i : \\ \qquad \langle s\_tot_k^{m_k}, k\rangle > \langle s\_tot_j^m, j\rangle \end{array}\right\}$
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**

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 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 $\rightsquigarrow_M$. 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 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.*

The proof, provided in the appendix relies on the monotonicity of the clocks $causal_i[1..n]$ and $total_i[1..n]$, and the reliability and FIFO properties of the underlying communication channels [7,12,23,34].

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

5.1 The high level object operations read and write

Algorithm 2 uses the $\mathcal{G}$-fisheye (SC,CC)-broadcast we have just presented to realize $\mathcal{G}$-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 [7], in which the total order broadcast has been replaced by our $\mathcal{G}$-fisheye (SC,CC)-broadcast.

The 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* write(X, v) by p_i are properly processed, p_i enters a waiting loop (line 4), which ends after the message 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 WRITE(X, v, j) messages are toco-delivered.

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
```

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

The proof (cf. appendix) 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 $\overset{\star}{\rightsquigarrow}_{H,\mathcal{G}}$ by adapting the technique used in [28,32].

6 Conclusion

This work was motivated by the increasing popularity of geographically distributed systems. We have presented a framework that enables to formally define and reason about mixed consistency conditions in which the operations invoked by nearby processes obey stronger consistency requirements than operations invoked by remote ones. The framework is based on the concept of a proximity graph, which captures the "closeness" relationship between processes. As an example, we have formally defined $\mathcal{G}$-fisheye (SC,CC)-consistency, which combines sequential consistency for operations by close processes with causal consistency among all operations. We have also provided a formally proven protocol for implementing $\mathcal{G}$-fisheye (SC,CC)-consistency.

Another natural example that has been omitted from this paper for brevity is $\mathcal{G}$-fisheye (LIN,SC)-consistency, which combines linearizability for operations by nearby nodes with an overall sequential consistency guarantee.

The significance of our approach is that the definitions of consistency conditions are functional rather than operational. That is, they are independent of a specific implementation, and provide a clear rigorous understanding of the provided semantics. This formal underpinning comes with improved complexity and performance, as illustrated in our implementation of $\mathcal{G}$-fisheye (SC,CC)-consistency, in which operations can terminate without waiting to synchronize with remote parts of the system.

More generally, we expect the general philosophy we have presented to extend to Convergent Replicated Datatypes (CRDT) in which not all operations are commutative [29]. These CRDTs usually require at a minimum causal communications to implement eventual consistency. The hybridization we have proposed opens up the path of CRDTs

which are globally eventually consistent, and locally sequentially consistent, a route we plan to explore in future work.

Acknowledgments

This work was partially funded by the French ANR project SocioPlug (ANR-13-INFR-0003), and by the DeSceNt project (Labex CominLabs excellence laboratory ANR-10-LABX-07-01).

References

1. Adve S. and Gharachorloo K., Shared memory consistency models: a tutorial. *IEEE Comp. Mag.*, 29(12):66-76, 1996.
2. Ahamad M., Niger G., Burns J.E., Hut to P.W., and Kohl P., Causal memory: definitions, implementation and programming. *Dist. Computing*, 9:37-49, 1995.
3. Almeida S., Leitaõ J., Rodrigues L., ChainReaction: a Causal+ Consistent Datastore based on Chain Replication. *8th ACM Europ. Conf. on Comp. Sys. (EuroSys'13)*, pp. 85-98, 2013.
4. Alvaro P., Bailis P., Conway N., and Hellerstein J. M., Consistency without borders *4th ACM Symp. on Cloud Computing (SOCC '13)*, 2013, 23
5. Attiya H. and Friedman R., A correctness condition for high-performance multiprocessors. *SIAM Journal on Computing*, 27(6):1637-1670, 1998.
6. Attiya H. and Friedman R., Limitations of Fast Consistency Conditions for Distributed Shared Memories. *Information Processing Letters*, 57(5):243-248, 1996.
7. Attiya H. and Welch J.L., Sequential consistency versus linearizability. *ACM Trans. on Comp. Sys.*, 12(2):91-12, 1994.
8. Attiya H. and Welch J.L., *Distributed computing: fundamentals, simulations and advanced topics, (2nd Edition)*, Wiley-Inter science, 414 pages, 2004 (ISBN 0-471-45324-2).
9. Bailis P., Ghodsi A., Hellerstein J. M., and Stoica I., Bolt-on Causal Consistency *2013 ACM SIGMOD Int. Conf. on Management of Data (SIGMOD'13)*, pp. 761-772, 2013.
10. Birman K.P. and Friedman R., Trading Consistency for Availability in Distributed Systems. *Technical Report #TR96-1579*, Computer Science Department, Cornell University, April 2016.
11. Birman K.P. and Joseph T.A., Reliable communication in presence of failures. *ACM Trans. on Comp. Sys.*, 5(1):47-76, 1987.
12. Birman K., Schiper A., and Stephenson P., Lightweight Causal and Atomic Group Multicast *ACM Trans. Comput. Syst.*, vol. 9, pp. 272-314, 1991.
13. Brewer E., Towards Robust Towards Robust Distributed Systems *19th ACM Symposium on Principles of Distributed Computing (PODC)*, Invited talk, 2000.
14. Burckhardt S., Gotsman A., Yang H., and Zawirski M., Replicated Data Types: Specification, Verification, Optimality *41st ACM Symp. Princ. of Prog. Lang. (POPL)*, pp. 271-284, 2014.
15. DeCandia G., Hastorun D., Jampani M., Kakulapati G., Lakshman A., Pilchin A., Sivasubramanian S., Vosshall P., and Vogels W., Dynamo: amazon's highly available key-value store *21st ACM Symp. on Op. Sys. Principles (SOSP'07)*, pp. 205-220, 2007
16. Friedman R., Implementing Hybrid Consistency with High-Level Synchronization Operations. *Distributed Computing*, 9(3):119-129, 1995.
17. Garg V.K. and Raynal M., Normality: a consistency condition for concurrent objects. *Parallel Processing Letters*, 9(1):123-134, 1999.
18. Gharachorloo K., Lenoski D., Laudon J., Gibbons P., Gupta A., and Hennessy J., Memory consistency and event ordering in scalable shared-memory multiprocessors. *17th ACM Annual International Symp. on Comp. Arch. (ISCA)*, pp. 15-26, 1990.

19. Herlihy M. and Shavit N., *The Art of Multiprocessor Programming*, Morgan Kaufmann Publishers Inc., 508 pages, 2008 (ISBN 978-0-12-370591-4).
20. Herlihy M. and Wing J., Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.
21. Keleher P. Cox A.L., and Zwaenepoel W., Lazy release consistency for software distributed shared memory. *Proc. 19th ACM Int'l Symp. on Comp. Arch. (ISCA'92)*, pages 13–21, 1992.
22. Lakshman A., and Malik P., Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, volume 44, pp. 35-40, 201.
23. Lamport L., Time, Clocks and the Ordering of Events in a Distributed System *Comm. of the ACM*, vol. 21, pp. 558-565, 1978
24. Lamport L., How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. on Comp.*, C28(9):690–691, 1979.
25. PRAM: A Scalable Shared Memory. *Technical Report CS-TR-180-88*, Princeton University, September 1988.
26. Lloyd W., Freedman M. J., Kaminsky M., and Andersen D. G., Don't Settle for Eventual: Scalable Causal Consistency for Wide-area Storage with COPS. *23rd ACM Symp. on Op. Sys. Principles*, pp. 401-416, 2011.
27. Lynch N.A., *Distributed Algorithms.* Morgan Kaufman, San Francisco, 872 pages, 1996.
28. Mizuno M., Raynal M., and Zhou J. Z., Sequential Consistency in Distributed Systems. *Selec.Papers from the Int. W. on Theory & Prac. in Dist. Sys.*, Springer, pp. 224-241, 1995
29. Oster G., Urso P., Molli P., and Imine A., Data consistency for P2P collaborative editing. *20th Anniv. Conf. on Comp. Supported Cooperative Work*, ACM, pp. 259-268, 2006
30. Preguiça N.M., Marquès J.M., Shapiro M., and Letia M., A Commutative Replicated Data Type for Cooperative Editing. *Proc. 29th IEEE Int'l Conf. on Dist. Comp. Sys. (ICDCS'09)*, pp. 395–403, 2009.
31. Raynal M., *Concurrent Programming: Algorithms, Principles, and Foundations*, Springer, 515 pages, 2013, ISBN 978-3-642-32026-2.
32. Raynal M., *Distributed Algorirhms for Message-passing Systems*, Springer, 500 pages, 2013, ISBN 978-3-642-38122-5.
33. Raynal M. and Schiper A., A suite of formal definitions for consistency criteria in distributed shared Memories. *9th Int'l IEEE Conf. on Parallel and Dist. Comp. Sys. (PDCS'96)*, pp. 125-131, 1996.
34. Raynal M., Schiper A., and Toueg S., The Causal Ordering Abstraction and a Simple Way to Implement. *Information Processing Letters*, 39(6):343-350, 1991.
35. Saito Y. and Shapiro M., Optimistic Replication. it ACM Computing Survey, 37(1):42-81, March 2005.
36. Shapiro M., Preguiça N.M., Baquero C., and Zawirski M., Convergent and Commutative Replicated Data Types. *Bulletin of the EATCS*, 104:67-88, 2011.
37. Sovran Y., Power R., Aguilera M. K., and Li J., Transactional Storage for Geo-Replicated Systems. *23rd ACM Symp. on Operating Sys. Princ. (SOSP)*, pp. 385-400, 2011.
38. Terry D. B., Prabhakaran V., Kotla R., Balakrishnan M., Aguilera M. K., and Abu-Libdeh H., Consistency-based Service Level Agreements for Cloud Storage, *24th ACM Symp. on Op. Sys. Principles (SOSP'13)*, pp. 309-324, 2013.
39. Terry D. B., Theimer M. M., Petersen K., Demers A. J., Spreitzer M. J., and Hauser C. H., Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System *15th ACM Symp. on Op. Sys. Principles (SOSP'95)*, pp. 172-182, 1995.
40. Xie C., Su C., Kapritsos M., Wang Y., Yaghmazadeh N., Alvisi L., and Mahajan P., SALT: Combining ACID and BASE in a Distributed Database. *USENIX Op. Sys. Design & Imp. (OSDI)*, 2014.

Appendix: Detail of Proofs

Proof that Algorithm 1 implements a $\mathcal{G}$-fisheye (SC,CC)-broadcast

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:

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

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 3. *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:

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

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 $\langle s\_tot_k^{m_k}, k\rangle \leq \langle s\_tot_i^{m}, i\rangle$ 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 timestamp 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. $\square_{Theorem\ 3}$

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):

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

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-broadcast 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 4. *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 toco-delivered m_1. This follows from the transitivity of temporal order. This result together with Lemmas 4 and 5 concludes the proof. $\square_{Theorem\ 4}$

$\mathcal{G}$-delivery order

Theorem 5. *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 18-31) 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:

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

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. $\square_{Theorem\ 5}$

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

Proof

- Validity and Integrity follow from the validity and integrity 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 5;
– Causal order follows from Theorem 4;
– Termination follows from Theorem 3.

$\square_{Theorem\ 1}$

Proof that Algorithm 2 implements $\mathcal{G}$-fisheye (SC,CC)-consistency

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 $\rightarrow v$), 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 $\widehat{H} = (H, \overset{po}{\rightarrow}_H)$ that captures an execution of Algorithm 2, i.e., $\overset{po}{\rightarrow}_H$ 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 3.2 in the following, classical, manner:

– We connect each read operation $r_p(X,v) = X$.read $\rightarrow v$ invoked by process p (with $v \neq \perp$, the initial value) to the write operation $w(X,v) = X$.write(v) that generated the WRITE(X,v) message carrying the value v to p (line 10 in Algorithm 2). In other words, we add an edge $\langle w(X,v) \overset{rf}{\rightarrow} r_p(X,v)\rangle$ to $\overset{po}{\rightarrow}_H$ (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 $\perp$. We connect initial read operations $r(X,\perp)$ to an $\perp$ element that we add to H.
We call these additional relations *read-from links* (noted $\overset{rf}{\rightarrow}$).
– 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 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 in the sense of the definition of Section 2.3, 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 $\rightsquigarrow_H$ (said differently, that no read returns an overwritten value).

Lemma 6. *$\rightsquigarrow_H$ 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 $w_p(X,v) \overset{rf}{\rightarrow} r_q(X,v)$, which implies $w_p(X,v) \rightsquigarrow_H r_q(X,v)$. Let us assume there exists a second write operation $w_r(X,v') \neq w_p(X,v)$ on the same object, so that

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

(illustrated in Figure 7). $w_p(X,v) \rightsquigarrow_H w_r(X,v')$ means we can find a sequence of operations $op_i \in H$ so that

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

with $\rightarrow_i \in \{\overset{po}{\rightarrow}_H, \overset{rf}{\rightarrow}\}, \forall i \in [1,k]$. The semantics of $\overset{po}{\rightarrow}_H$ and $\overset{rf}{\rightarrow}$ 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:

$$\text{WRITE}_p(X,v) = m_0 \rightsquigarrow_M m_1 ... \rightsquigarrow_M m_i \rightsquigarrow_M ... \rightsquigarrow_M m_{k'} = \text{WRITE}_r(X,v') \quad (36)$$

where $\rightsquigarrow_M$ is the message causal order introduced in Section 4.1. We conclude that $\text{WRITE}_p(X,v) \rightsquigarrow_M \text{WRITE}_r(X,v')$, 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 7 toco-delivers $\text{WRITE}_r(X,v')$ after $\text{WRITE}_p(X,v)$.

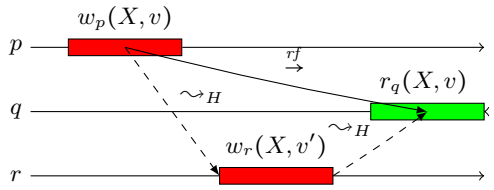

Fig. 7. Proving that $\rightsquigarrow_H$ is causal by contradiction

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 $w_r(X,v') \rightsquigarrow_H r_q(X,v)$, yielding another sequence of operations $op'_i \in H$:

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

with $\rightarrow'_i \in \{\overset{po}{\rightarrow}_H, \overset{rf}{\rightarrow}\}$. 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 $\overset{po}{\rightarrow}_H$ (i.e., $op'_i = \overset{po}{\rightarrow}_H, \forall i$). In this case, $r = q$, and the blocking behavior of X.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 $\text{WRITE}_r(X,v') \rightsquigarrow_M \text{WRITE}_z(Y,w)$. 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 $w_p(X,v) \overset{rf}{\rightarrow} r_q(X,v)$, and concluding our proof that $\rightsquigarrow_H$ 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 3.2), 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 $w_p(X,v) \overset{ww}{\rightarrow} w_q(Y,w)$ 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 $\overset{ww}{\rightarrow}$).
– 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 4.1), 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 $\overset{\mathcal{G}}{\rightsquigarrow}_M$:
$m_1 \overset{\mathcal{G}}{\rightsquigarrow}_M m_2$ 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 $m_1 \overset{\mathcal{G}}{\rightsquigarrow}_M m_3$ and $m_3 \overset{\mathcal{G}}{\rightsquigarrow}_M m_2$.

$\overset{\mathcal{G}}{\rightsquigarrow}_M$ 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 $\overset{\mathcal{G}}{\rightsquigarrow}_M$, instead of $\rightsquigarrow_M$. $\square_{Lemma\ 7}$

Theorem 2. *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 3.2):

– 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 $(\overset{\star}{\rightsquigarrow}_{H,\mathcal{G}})|(p_i + W)$, following the approach of [28,32]. $\widehat{S_i}$ is sequential by construction. Because $\overset{\star}{\rightsquigarrow}_{H,\mathcal{G}}$ is causal, $\widehat{S_i}$ is legal. Because $\overset{\star}{\rightsquigarrow}_{H,\mathcal{G}}$ respects $\overset{po}{\rightarrow}_H$, $\widehat{S_i}$ is equivalent to $\widehat{H}|(p_i + W)$. Finally, $\widehat{S_i}$ respects $(\overset{\star}{\rightsquigarrow}_{H,\mathcal{G}})|(p_i + W)$ by construction. $\square_{Theorem\ 2}$