Total Order Multicast to Multiple Groups

André Schiper Rachid Guerraoui
Département d'Informatique
Ecole Polytechnique Fédérale de Lausanne
1015 Lausanne, Switzerland

Abstract

We present a fault-tolerant algorithm which ensures *global total order* delivery of messages sent to multiple groups of processes. Our algorithm is a multiple group "genuine" multicast algorithm in the sense that (1) any process can send a message to any set of process groups, and (2) only the sender and the receivers of a message take part in the algorithm needed to deliver the message. The correctness of our algorithm does not require reliable failure detectors, but requires causal order delivery of messages. This establishes an interesting link between causal order delivery and fault-tolerance with unreliable failure detectors.

1 Introduction

We present in the paper a fault-tolerant total order multicast algorithm that allows to multicast messages to multiple groups. Before discussing the algorithm, we start by clarifying some issues related to the models that have been considered for total order multicast communication. We consider the distinctions *open* vs *closed* group, *global* vs *local* total order, and *genuine* vs *feigned* multicast.

1.1 Total order multicast models

Open groups vs closed groups. Systems that support group communications can be divided into two categories: (1) systems with *open groups*, and (2) systems with *closed groups* [23]. In a system with open groups, any process p_i can multicast a message to any group g, whereas in a system with closed groups, a process p_i can multicast a message m to a group g only if p_i is a member of g. Closed groups have been introduced for implementation reasons [23]. From a user perspective, open groups are obviously more general. Consider for instance a group g of replicated servers: with open groups, a client process p_i is allowed to issue a multicast to g without being member of g. To achieve the same result with closed groups, one has to create a group g' including the client p_i *and* the set of replicated servers g.

Global total order vs local total order. Total order multicast algorithms can be classified according to the strength of their ordering properties, which leads to distinguish between *local* and *global* total order [14]:

- *Local* total order ensures that two messages m_1 and m_2 multicast to the *same* destination set (e.g. to the same group g) are delivered in the same order by all the processes in the destination set.

  Local total order does not, however, order two messages multicast to two different destination sets. For instance, let m_1 be a message multicast to $\mathsf{Dst}(m_1)$ and m_2 be a message multicast to $\mathsf{Dst}(m_2)$. Two processes $p_j, p_k \in \mathsf{Dst}(m_1) \cap \mathsf{Dst}(m_2)$ might deliver m_1 and m_2 in a different order (p_j might deliver m_1 before m_2, whereas p_k might deliver m_2 before m_1).

- *Global* total order ensures that: (1) there is a total order on all the messages that are multicast in the system, and (2) the messages are delivered according to this order. Global total order defines a stronger property than local total order.

Genuine vs feigned multicast. By defining a single group G including all the processes in the system, one might indeed ensure *global* total order using *local* total order. The multicast of a message m to a destination set $\mathsf{Dst}(m) \subset G$ can be implemented as follows: (1) multicast m to G with local total order guarantee, and (2) when m is about to be delivered at some process p_i according to the local total order algorithm, then deliver m if and only if $p_i \in \mathsf{Dst}(m)$. We call this a "feigned" total order multicast algorithm, as it involves the cost of a total order *broadcast* algorithm. The solution is obviously not scalable, and not realistic in a system with a large number of processes.

We define a global total order algorithm to be a *"genuine"* multicast algorithm, if the following condition is satisfied: *the execution of the algorithm implementing the multicast of* *a message m to a destination set* $\mathsf{Dst}(m)$*, involves only the sender process, and the processes in* $\mathsf{Dst}(m)$. The above feigned multicast algorithm does not satisfy this property, whatever total order broadcast algorithm is used.

Whereas a fault-tolerant total order *broadcast* algorithm relying on an unreliable failure detector has been described in the literature (Chandra-Toueg's algorithm [5]), the only total order genuine multicast algorithms that we know of, either do not tolerate failures (e.g [21]), or assume a reliable failure detector (e.g [11, 15, 13]), i.e. a failure detector that never makes any incorrect failure suspicion. We know of no total order genuine multicast algorithm that does not require a reliable failure detector.

1.2 Multicast to multiple groups

In this paper, we consider the open group model, and we address the problem of global total order "genuine" multicast to *multiple groups*: we note $TOCAST(m, \{g_1, \ldots, g_n\})$ the primitive that multicasts the message m to the groups $g_1, \ldots, g_n$, with global total order delivery guarantee.

To illustrate the use of such a primitive, consider the following example. Let g_1 be the group of replicas that manage a bank account #1, and g_2 the group of replicas that manage a bank account #2. Consider the transfer of some amount A from bank account #1 to bank account #2. The two operations (withdrawal from account #1 and deposit to account #2) can be aggregated into a single message m by defining m as: *(remove A from account #1, deposit A to account #2)*. Thus the money transfer can be handled by the primitive $TOCAST(m, \{g_1, g_2\})$: when a process in g_1 delivers m, it removes A from the bank account it manages, and when a process in g_2 delivers m, it adds A to the bank account it manages. The money transfer from account #1 to account #2 can be viewed as a transaction, and the global total order property of the TOCAST primitive ensures the serializability property of transactions (for an extensive discussion of the link between a TOCAST primitive to multiple groups and transactions, see [20]).

The algorithm we propose, named *MTO* (Multicast Total Order), is a fault-tolerant extension of Skeen's total order multicast algorithm described in [21] [1]. The correctness of our MTO algorithm is based on the three following assumptions: (1) the failure detector is of

[1] Skeen's original algorithm is not fault-tolerant.

the class $\diamond\mathcal{S}$ [5], (2) in each group there is a majority of correct processes, and (3) causal order delivery is ensured. Failure detectors of the class $\diamond\mathcal{S}$ are those which ensure that (1) eventually every process that crashes is suspected by all correct processes, and (2) eventually some correct process is never suspected by any correct process. Such failure detectors are unreliable in the sense that they can make an infinity of false failure suspicions; it has been shown that such failure detectors are the weakest one that allow to solve consensus and total order broadcast problems [4]. Assumption (2) turns out to be necessary for any total order genuine multicast algorithm based on unreliable failure detectors [13]. Assumption (3) is actually not needed if failure detectors are reliable. This establishes an interesting link between causal order delivery and fault-tolerance with unreliable failure detectors.

The rest of the paper is structured as follows. Section 2 is devoted to the presentation of related work. Section 3 gives the system model and defines global total order multicast. Section 4 recalls the original Skeen's algorithm, and describes the MTO algorithm. Section 5 analyses the cost of MTO, and points out the tradeoff between total order broadcast and total order multicast. Section 6 concludes the paper. The proofs are in the Appendix.

2 Related work

Total order algorithms are usually classified as assuming either a synchronous system model [8], or an asynchronous system model. We consider hereafter only the asynchronous system model [2]. Orthogonal to the synchronous/asynchronous classification, algorithms can be classified as being either *symmetric* or *asymmetric*: in a symmetric algorithm all processes perform the same code [2, 18, 5], whereas in an *asymmetric* algorithm one process plays a special role, i.e. defines the ordering of messages [16, 3]. Asymmetric algorithms require less phases and are thus more efficient, but are subject to the contamination problem [12, 24]. Token based algorithms [7, 1] can be classified somewhere in between symmetric and asymmetric algorithms. We consider in the paper total order algorithms from a different perspective, i.e. we consider the following issues: *open* vs *closed* group, *global* vs *local* total order, and *genuine* vs *feigned* multicast.

The Isis ABCAST algorithm described in [3] ensures only local total order: it does not

[2] More precisely, we consider an asynchronous system augmented with unreliable failure detectors.

order messages in intersecting groups. Stephenson [22] tries to address the intersecting group issue, and proposes a solution close to the first Isis ABCAST algorithm [2]. The first Isis ABCAST algorithm is based on Skeen's algorithm [21] and has in that sense similarities with the MTO described in this paper. However, fault-tolerance is ensured in the Isis algorithm by a membership service, whereas MTO does not require a membership service. Moreover, because of the membership service of the Isis algorithm, the need for causal order that we discuss in this paper has not been identified. Also, Isis is based on the closed group model.

The algorithms based on Lamport's total order algorithm, i.e. the algorithms which deliver the messages according to the order defined by the time-stamps initially assigned to messages at multicast time [9, 1, 10], cannot satisfy the genuine multicast property defined in Section 1. Indeed, a process p can deliver a message m, time-stamped $ts(m)$, only once p knows that it will receive no further message m' such that $ts(m') < ts(m)$. This might require p to interact with all the processes that can send a message to p. In the case where p can receive a message from any process in the system, p might need to interact with the whole system. One way to circumvent the problem is to restrict the processes from which messages can be received by considering the closed group model [10]. The constraining closed group model does not exist in the MTO algorithm given in the paper. Moreover, when a process p_i multicasts a message m to a set of destination processes $\mathsf{Dst}(m)$, the MTO algorithm requires only the participation of p_i and $\mathsf{Dst}(m)$: MTO is thus a genuine multicast algorithm.

Finally the algorithm presented by Garcia-Molina and Spauster [11], and its extension described in [15], are both total order genuine multicast algorithms but both require reliable failure detectors, i.e. failure detectors that do not make false suspicions. The only total order algorithm that relies on unreliable failure detectors is the *broadcast* algorithm by Chandra and Toueg [5]. To our knowledge, MTO is the only global total order genuine multicast algorithm based on unreliable failure detectors.

3 System model and definitions

3.1 Asynchronous system

We consider a distributed system composed of a finite set of processes $\Omega = \{p_1, p_2, \ldots, p_n\}$ completely connected through a set of channels. Communication is by message passing, asynchronous and reliable. Asynchrony means that there is no bound on communication delays, nor on process relative speeds. A reliable channel ensures that a message sent by a process p_i to a process p_j is eventually received by p_j, if p_i and p_j are correct (i.e. do not fail)[3]. A reliable channel can be implemented by retransmitting lost or corrupt messages. Processes fail by crashing; we do not consider Byzantine failures. A process $p_i \in \Omega$ may (1) send a message to another process (*send* event), (2) receive a message sent by another process (*receive* event), (3) perform some local computation, or (4) fail.

3.2 Failure detectors

As in [5], we augment the asynchronous system with failure detectors. A failure detector can be viewed as a distributed oracle. Each process p_i has access to a local failure detector module D_i. This module maintains a list of processes that it currently *suspects* to have crashed. We do not make any assumption on how a failure detector is implemented. It can, for example, simply be based on time-outs, and failure detector modules can consult each other before suspecting a process.

Failure detectors are abstractly characterized by their *completeness* and *accuracy* properties. In the following, we consider failure detectors in class $\diamond\mathcal{S}$ (*Eventually Strong*) [5]. These failure detectors guarantee (1) *strong completeness* (eventually every process that crashes is permanently suspected by every correct process), and (2) *eventual weak accuracy* (eventually some correct process is never suspected by any correct process). Notice that failure detectors in $\diamond\mathcal{S}$ are unreliable: they can make mistakes, i.e false failure suspicions. The fundamental characteristic of $\diamond\mathcal{S}$ is that it is the *weakest* class of failure detectors that allows to solve the consensus and the atomic broadcast problems [5].

3.3 Total order multicast to multiple groups

We assume that the set Ω of processes is partitioned into groups g_i, and messages are multicast to groups. Given a message m, we note $\mathsf{Dst}(m)$ the set of process groups to which m is multicast. Total order multicast is defined on the set Ω by (1) the primitive *TO-multicast*$(m, \mathsf{Dst}(m))$ which issues m to $\mathsf{Dst}(m)$, and (2) *TO-deliver(m)* which is the corresponding delivery of m. When a process p_i executes *TO-multicast(m)* (resp *TO-deliver(m)*),

[3] This does not exclude link failures, if we require that any link failure is eventually repaired.

we say that p_i *"TO-multicasts m"* (resp *"TO-delivers m"*). The primitive *TO-multicast* is defined by the following three properties [14]:

TO1. Global total order. Let m_1 and m_2 be two messages that are *TO-multicast*. We note $m_1 < m_2$ if and only if a process (correct or not) *TO-delivers* m_1 before m_2. Global total order ensures that the relation $<$ is acyclic.

TO2. Uniform Agreement. Consider *TO-multicast*$(m, \mathsf{Dst}(m))$. If a process in $\mathsf{Dst}(m)$ (correct or not) has *TO-delivered*(m), then every correct process in $\mathsf{Dst}(m)$ eventually *TO-delivers*(m).

TO3. Validity. If a correct process *TO-multicasts*$(m, \mathsf{Dst}(m))$, then every correct process in $\mathsf{Dst}(m)$ eventually *TO-delivers*(m).

3.4 Implementation model

We introduce below the implementation model that is needed to describe our MTO algorithm. These assumptions are discussed in Section 4.6.

Lamport's clocks. Given an event e, we note $ts(e)$ its time-stamp using Lamport's logical clocks [17] (as usual, we consider that the process ID is added as a suffix to the logical clock values, to ensure that all the time-stamps are unique). Also, given two events e_1 and e_2, we note $e_1 \rightarrow e_2$ the happened-before relation defined by Lamport [17].

Causal order delivery. We assume causal order delivery for all the messages sent in the system, and we note $deliver_i(m)$ the delivery of message m at process p_i according to causal order. Causal order delivery ensures that given two messages m and m' sent to a common destination p_i, if $send(m) \rightarrow send(m')$, then $deliver_i(m) \rightarrow deliver_i(m')$. We do not discuss here implementation of causal order delivery (see [19] for example).

Reliable multicast. To give a modular presentation of our MTO algorithm, we assume the existence of a *reliable* multicast primitive, noted *R-multicast(m,*$\mathsf{Dst}(m)$*)*, and the corresponding reliable delivery primitive, noted *R-deliver(m)*. The primitive *R-multicast(m,*$\mathsf{Dst}(m)$*)* is defined by the following two properties: (1) if a process in $\mathsf{Dst}(m)$ (correct or not) has

R-delivered(m), then every correct process in $\mathsf{Dst}(m)$ eventually *R-delivers*(m); (2) if a correct process *R-multicasts*$(m,\mathsf{Dst}(m))$, then every correct process in $\mathsf{Dst}(m)$ eventually *R-delivers*(m) [14]. *R-multicast*$(m,\mathsf{Dst}(m))$ can be implemented by having each process in $\mathsf{Dst}(m)$ forwarding m to $\mathsf{Dst}(m)$ before delivering m [5]. We consider reliable multicast implemented on top of the causal communication layer defined above. Thus reliable multicast satisfies causal delivery order. Moreover, we assume that causal order delivery is also ensured between (1) messages issued with the *send* primitive, and (2) messages issued with the *R-multicast* primitive.

Qualified majority. We note f_i the maximum number of processes of group g_i that can crash, and we assume $f_i < |g_i|/2$ for every group g_i, i.e. we assume a majority of correct processes in each group. Given *TO-multicast*$(m, \mathsf{Dst}(m))$, we define the *qualified majority* of $\mathsf{Dst}(m)$ as a set of processes $QM(\mathsf{Dst}(m))$, such that $QM(\mathsf{Dst}(m))$ contains a majority of processes of every group g_i in $\mathsf{Dst}(m)$.

As an example, if $\mathsf{Dst}(m) = \{g_1, g_2\}$, $g_1 = \{p_1, p_2, p_3\}$ and $g_2 = \{p_4, p_5, p_6\}$, then $\{p_1, p_2, p_4, p_5\}$ is a qualified majority of $\mathsf{Dst}(m)$, but not $\{p_1, p_2, p_3, p_4\}$.

4 The MTO algorithm

4.1 Skeen's original algorithm

As MTO can be viewed as a fault-tolerant extension of Skeen's total algorithm [21], we first recall the latter. We make here the assumption that no process can be subject to crash failures (Skeen's algorithm does not tolerate failures). Figure 1 shows the three parts of the algorithm on a simple example of a process p_i *TO-multicasting* a message m to two groups g_1 and g_2, i.e. $\mathsf{Dst}(m) = \{g_1, g_2\}$. Skeen's algorithm works as follows [4]:

TO-multicast. When *TO-multicast*$(m, \mathsf{Dst}(m))$ is executed by process p_i, the message m is sent to every process in $\mathsf{Dst}(m)$ (part 1 in Fig. 1).

TO-delivery. The TO-delivery of m occurs as the result of the following sequence of actions:

[4] Skeen's algorithm does not require causal order delivery. Thus we use *send*(m) and *receive*(m) when describing Skeen's algorithm, rather than *send*(m) and *deliver*(m).

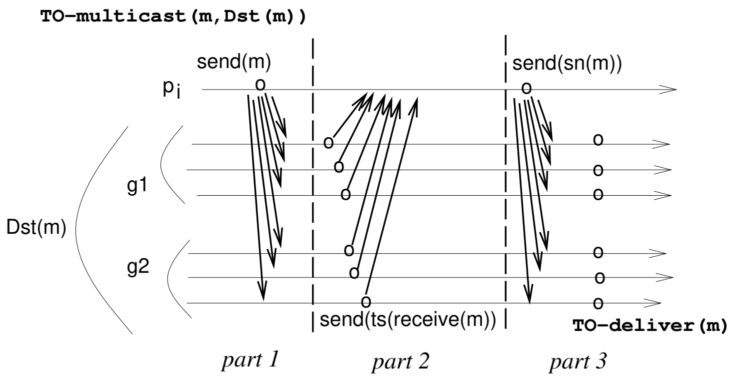

Figure 1: Skeen's original algorithm

- when $p_j \in \mathsf{Dst}(m)$ receives m, it sends $ts(receive_j(m))$ (the time-stamp of the $receive_j(m)$ event) back to p_i (part 2 in Fig. 1). Process p_j then stores the pair $(m, ts(receive_j(m))$ in a *"pending"* buffer;

- the initiator p_i waits to obtain the time-stamp $ts(receive_j(m))$ from every process p_j in $\mathsf{Dst}(m)$, and defines $sn(m)$ as the maximum of all time-stamps. Process p_i then sends $sn(m)$ to $\mathsf{Dst}(m)$ (part 3 in Fig. 1);

- when $p_j \in \mathsf{Dst}(m)$ receives $sn(m)$, the pair $(m, ts(receive_j(m))$ is removed from p_j's the pending buffer, and $(m, sn(m))$ is inserted into p_j's *"TO-delivery"* queue, ordered according to $sn(m)$ (the pair $(m, sn(m))$ with the smallest $sn(m)$ is at the head of the queue). The message m at the head of the TO-delivery queue is TO-delivered by p_j if and only if there is no pair $(m', ts(receive_j(m'))$ in p_j's pending buffer, such that $ts(receive_j(m')) < sn(m)$.

4.2 The structure of the MTO algorithm

The MTO algorithm can be viewed as a fault-tolerant extension of Skeen's algorithm. The principle of both algorithms is similar in that both order messages according a sequence number that is computed after the message has been sent. The specificities of MTO compared to Skeen's original algorithm are: (1) a reliable multicast (*R-Multicast*, see Sect. 3.4) is used to issue the message (part 1 in Fig. 2), (2) the sequence number is computed using a specific *consensus* function, noted *MTO-SN* (part 2 in Fig. 2), and (3) causal delivery of messages is required. The MTO algorithm works as follows:

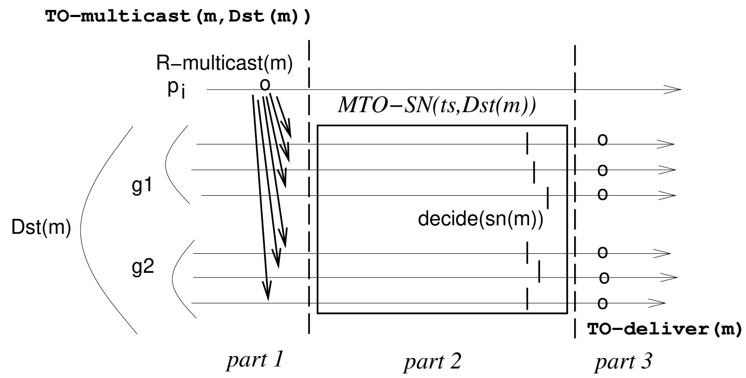

Figure 2: Principle of the MTO algorithm

TO-multicast. When *TO-multicast*$(m, \mathsf{Dst}(m))$ is executed by a process p_i, the message m is *R-multicast* to $\mathsf{Dst}(m)$.

TO-delivery. The TO-delivery of m occurs as the result of the following sequence of actions:

1. consider $p_j \in \mathsf{Dst}(m)$ and let $ts(R\text{-}deliver_j(m))$ be the time-stamp of the $R\text{-}deliver_j(m)$ event. Process p_j stores the pair $(m, ts(R\text{-}deliver_j(m))$ in its *pending* buffer;

2. p_j then computes $sn(m)$ by calling the function *MTO-SN* with $ts(R\text{-}deliver_j(m))$ and $\mathsf{Dst}(m))$ as parameters (part 2 in Fig. 2 and Sect. 4.3);

3. when the call to *MTO-SN* terminates, $sn(m)$ is known by p_j. At that point (part 3 in Fig. 2), the pair $(m, ts(R\text{-}deliver_j(m))$ is removed from p_j's pending buffer and is inserted into p_j's TO-delivery queue, ordered according to $sn(m)$ (the pair with the smallest $sn(m)$ at the head of the queue). The message m at the head of the TO-delivery queue is finally TO-delivered by p_j if and only if there is no pair $(m', ts(R\text{-}deliver_j(m'))$ in p_j's pending buffer, such that $ts(R\text{-}deliver_j(m')) < sn(m)$.

The next section describes the *MTO-SN* function. Section 4.4 shows that the properties TO2 (Uniform Agreement) and TO3 (Validity) are satisfied. Section 4.5 finally shows that causal order delivery ensures the ordering property TO1.

4.3 The MTO-SN function: computation of the sequence number $sn(m)$

The function MTO-SN computing $sn(m)$ is given in Figure 3. When MTO-SN, called by some process p_j, returns (line 6) we say that $sn(m)$ is "*decided*" by p_j. At line 5, MTO-SN calls

```
function MTO-SN (ts_j, Dst(m))

    send ts_j to Dst(m) ;
    wait until [delivered ts_k from a qualified majority of Dst(m)] ;
    prop_j ← max of all ts_k delivered ;
    res_j ← QM-unif-consensus (prop_j, Dst(m)) ;
    return res_j ;
```

Figure 3: Computing the sequence number $sn(m)$: code of process p_j

a function named *QM-unif-consensus*. This function solves a simple variation of the uniform consensus problem[5]. It can be shown that any algorithm that solves uniform consensus among a set Ω of processes using a failure detector of class $\diamond\mathcal{S}$ is such that if the decision v is taken at time t, then a "majority" of processes of Ω knows v at time t [6].

We define a "QM-unif-consensus" algorithm to be a uniform consensus algorithm such that when the decision v is taken at time t, a "qualified" majority (see Sect. 3.4) knows v at time t.

A QM-unif-consensus algorithm is trivially obtained from the Chandra-Toueg uniform consensus algorithm using $\diamond\mathcal{S}$ [5]. The transformation consists in replacing every "majority" condition in the algorithm by a "qualified majority" condition. The need for a QM-unif-consensus algorithm appears in Sect. 4.5.

4.4 MTO satisfies the properties TO2 (Uniform Agreement) and TO3 (Validity)

We show now that the MTO algorithm satisfies the properties TO2 and TO3. We start by two lemmas (proofs in the Appendix).

[5]The uniform consensus problem differs from the consensus problem by requiring "Uniform" Agreement (all processes, *correct or not*, decide the same value) instead of Agreement (all *correct* processes decide the same value). The Chandra-Toueg algorithm, using a failure detector of class $\diamond\mathcal{S}$, solves the uniform consensus problem [5].

[6]As processes learn through messages [6], the condition can be restated as follows. Let $decide_i(v)$ be the event by which p_i decides on v. Then there is a majority $MAJ \subseteq \Omega$, and a message m_v carrying v, such that for every process $p_j \in MAJ$ we have $deliver_j(m_v) \rightarrow decide_i(v)$.

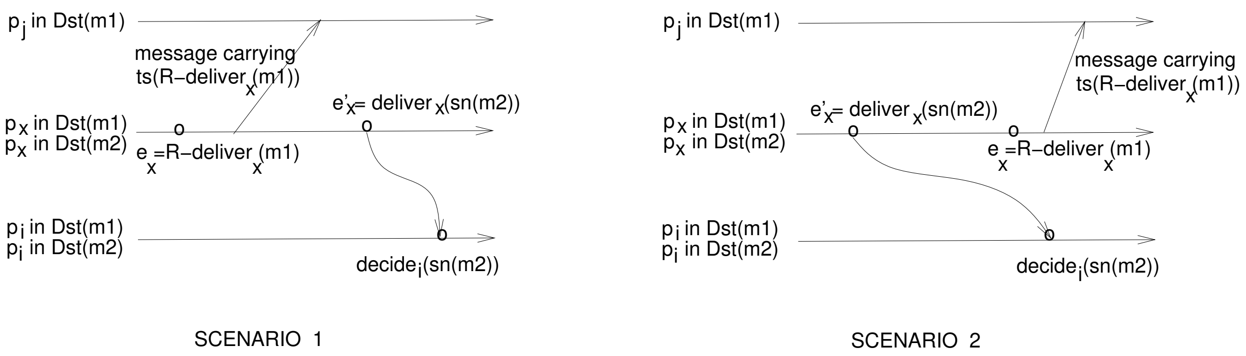

Figure 4: Illustration of the proof of Lemma 5.5 (the straight line between p_x and p_j represents a message, whereas the curved line between p_x and p_i represents a causal dependency).

Lemma 4.1 *Consider the execution of* TO-multicast$(m, \mathsf{Dst}(m))$. *If* $p_j \in \mathsf{Dst}(m)$ *is correct and R-delivers* m*, then every correct process in* $\mathsf{Dst}(m)$ *eventually decides* $sn(m)$.

Lemma 4.2 *Consider the execution of* TO-multicast$(m, \mathsf{Dst}(m))$ *by* p_i. *If* p_i *is correct, then every correct process in* $\mathsf{Dst}(m)$ *eventually decides* $sn(m)$.

Proposition 4.3 (TO2) *The MTO algorithm satisfies the Uniform Agreement property TO2.*

Proposition 4.4 (TO3) *The MTO algorithm satisfies the Validity property TO3.*

4.5 MTO satisfies the properties TO1 (Global Total Order)

To prove the correctness of the MTO algorithm, we still have to prove that the ordering property TO1 is satisfied. This is proved by Proposition 4.6 which requires Lemma 4.5.

Lemma 4.5 *Consider the two primitives TO-multicast*$(m_1, \mathsf{Dst}(m_1))$ *and TO-multicast*$(m_2, \mathsf{Dst}(m_2))$*, and assume* $sn(m_1) < sn(m_2)$. *If* p_i *is a process in* $\mathsf{Dst}(m_1) \cap \mathsf{Dst}(m_2)$*, then* p_i R-delivers m_1 *before* TO-delivering m_2.

The proof of the lemma is in the Appendix, and it is based on the two scenarios depicted in Figure 4. The figure shows the three key events, associated to every message m: (1) the R-delivery of m at p_x, noted *R-deliver*$_x(m)$, (2) the delivery at p_x of a message carrying the sequence number $sn(m)$, noted $deliver_x(sn(m))$ [7], and (3) the decision by p_i (through QM-unif-consensus) on the sequence number associated to m, noted $decide_i(sn(m))$. The events

[7] $deliver_x(sn(m))$ is the event after which p_x "knows about" $sn(m)$, although p_x has not yet decided on $sn(m)$.

(2) and (3) occur during the execution of the QM-unif-consensus algorithm that decides on $sn(m)$.

Consider *TO-multicast*$(m_1, \mathsf{Dst}(m_1))$ and *TO-multicast*$(m_2, \mathsf{Dst}(m_2))$ with $\mathsf{Dst}(m_1) \cap \mathsf{Dst}(m_2) \neq \emptyset$ (Lemma 4.5). There exists a process $p_x \in \mathsf{Dst}(m_1) \cap \mathsf{Dst}(m_2)$ (see Fig. 4) that plays a crucial role with respect to m_1 and m_2:

- the time-stamp of $R\text{-}deliver_x(m_1)$ is necessary to p_j in order to start the QM-unif-consensus that decides on $sn(m_1)$ (Fig 3, line 3, 4 and 5). Event e_x on p_x, in Figure 4, is $R\text{-}deliver_x(m_1)$;
- by the property of QM-unif-consensus (Sect. 4.3), the value $sn(m_2)$ is known to p_x, before p_i can decide on $sn(m_2)$. Event e'_x in Figure 4 is the delivery of the value $sn(m_2)$.

Figure 4 shows the two possible scenarios: either (1) $e_x \rightarrow e'_x$, or (2) $e'_x \rightarrow e_x$. In scenario 2, by the property of Lamport's clocks and because $sn(m_1)$ is defined as the maximum of delivery time-stamps, we get $sn(m_1) > sn(m_2)$, in contradiction with the assumption $sn(m_1) < sn(m_2)$ of Lemma 4.5. Therefore, scenario 1 is the only possible. In this case, we have $[e_x \equiv R\text{-}deliver_x(m_1)] \rightarrow e'_x \rightarrow decide_i(sn(m_2))$. Causal order delivery ensures that p_i R-delivers m_1 before delivering $sn(m_2)$, i.e. that p_i R-delivers m_1 before TO-delivering m_2, which completes the proof of Lemma 4.5.

Proposition 4.6 (TO1) *The MTO algorithm satisfies the ordering property TO1.*

4.6 On the necessity of the assumptions

To prove the correctness of our MTO algorithm, we have considered the following assumptions (Sect. 3): (1) the failure detector is of the class $\diamond\mathcal{S}$ [5], (2) in each group there is a majority of correct processes, and (3) causal order delivery is ensured. We discuss now the necessity of these assumptions.

It has been shown that, (i) failure detectors of class $\diamond\mathcal{S}$ are the weakest that allow to solve consensus and total order broadcast problems [4], and (ii) consensus and total order broadcast problems [4] cannot be solved with failure detectors of class $\diamond\mathcal{S}$ if less than a majority of processes can crash. As one can implement total order broadcast using total order multicast, it is obvious that failure detectors of class $\diamond\mathcal{S}$ are the weakest that allow to solve global total order multicast. Furthermore, since the destination set $\mathsf{Dst}(m)$ of a message m can be a single group g_i, solving global total order in g_i with $\diamond\mathcal{S}$ requires a majority of correct processes.

To illustrate the need for causal order delivery in MTO, consider the following example. Let m_1, m_2 be two messages such that $\mathsf{Dst}(m_1) = \mathsf{Dst}(m_2) = \{g\} = \{p_1, p_2, p_3\}$, and consider the two multicasts *TO-multicast*$(m_1, \mathsf{Dst}(m_1))$ (issued by some process p_k) and *TO-multicast*$(m_2, \mathsf{Dst}(m_2))$ (issued by some process p_l). Consider scenario 1 of Figure 4, with $p_j \equiv p_1$, $p_x \equiv p_x$ and $p_i \equiv p_3$. Without causal order delivery, we could have $sn(m_1) < sn(m_2)$, and m_2 TO-delivered by p_3, without p_3 having R-delivered m_1. In this case, either the Uniform Agreement property TO2 is violated, or the Global total order property TO1 is violated.

5 Cost analysis

In this section, we analyze the cost of the MTO algorithm in terms of number of messages and number of communication steps needed to TO-deliver a message. We compare these numbers with those of the Chandra-Toueg total order broadcast algorithm. The two algorithms are comparable in that both are based on unreliable failure detectors of the class $\diamond\mathcal{S}$ [5], and none is based on a membership service. We consider in our analysis only the best case scenario, i.e. runs with no failure and no failure suspicion. This is the most frequent case in practice.

Number of communication steps. [8] The MTO algorithm consists of (1) R-multicast of message m to $\mathsf{Dst}(m)$ (part 1 in Fig. 2), then (2) for every process p_j, the sending of ts_j to $\mathsf{Dst}(m)$ (line 2 in Fig. 3), and finally (3) the consensus. Let C be the number of communication steps needed to solve the consensus problem in the best case scenario. The MTO algorithm requires thus $2 + C$ communication steps. By comparison, the Chandra-Toueg total order broadcast algorithm consists of (1) R-broadcast of message m, followed by (2) the consensus. This leads to $1 + C$ communication steps.

[8] This measure is less ambiguous than the usual number of "phases". To give an example, the classical two phase commit protocol (2PC) has 3 communication steps: (1) *vote request* sent from the coordinator to the participants, (2) *reply* of the participants sent to the coordinator, and (3) *decision* sent by the coordinator to the participants.

	Communication steps	Messages
Chandra–Toueg (broadcast / unreliable FD)	4	$2(n-1)^2 + 2(n-1)$
MTO (multicast / unreliable FD)	5	$3(d-1)^2 + 4(d-1) + 1$

Figure 5: Trade off between total order broadcast / multicast ($d \leq n$)

For the Chandra-Toueg consensus algorithm based on $\diamond\mathcal{S}$, we have $C = 3$, which leads to 4 communication steps for the Chandra-Toueg total order broadcast algorithm in the best case, and 5 for the MTO algorithm.

Number of messages. Let $n = |\Omega|$ be the number of processes in the system, let m be a message TO-multicast to $\mathsf{Dst}(m)$ with $d = |\mathsf{Dst}(m)|$, and let M be the number of messages needed to solve the consensus problem in the best case scenario.

Consider first the MTO algorithm. If we assume that the initiator p_i of TO-multicast is not in $\mathsf{Dst}(m)$, then the initial R-multicast of m to $\mathsf{Dst}(m)$ costs $d + (d-1)^2$ messages (causal order delivery does not require any additional messages, but leads to larger messages [19]). For every process in $\mathsf{Dst}(m)$, sending ts_j to $\mathsf{Dst}(m)$ costs $d(d-1)$ messages. Thus MTO costs $M + 2(d-1)^2 + 2(d-1) + 1$ messages. By comparison, the Chandra-Toueg total order broadcast algorithm costs $(n-1)^2 + M$ messages (R-broadcast and consensus).

For Chandra-Toueg's consensus algorithm based on $\diamond\mathcal{S}$ among x processes, we have $M = 2(x-1) + (x-1)^2$ in the best case. This leads to a total of $3(d-1)^2 + 4(d-1) + 1$ messages for MTO in the best case, and $2(n-1)^2 + 2(n-1)$ messages for Chandra-Toueg's broadcast algorithm in the best case.

Trade-off. Figure 5 summarizes the numbers given above, for the case of a multicast sent to d processes in a system of n processes ($d \leq n$). All n processes take part to the protocol in the case of a broadcast, whereas only d processes take part to the protocol in the case of a multicast. The figure illustrates the interesting trade offs (1) between *broadcast* with 4 communication steps, where all n processes take part to the protocol, and (2) *multicast* with 5 communication steps, where only the d processes in $\mathsf{Dst}(m)$ take part to the protocol ($d \leq n$).

6 Discussion

The paper has presented a fault-tolerant total order multicast algorithm, named MTO, that allows to multicast messages to multiple groups of processes. The MTO algorithm considers an open group model, by allowing any process to send a message to any set of groups. MTO is a genuine multicast algorithm in the sense that, given *TO-multicast*$(m, \mathsf{Dst}(m))$, only the sender and the processes in $\mathsf{Dst}(m)$ take part in the protocol needed to TO-deliver m. The only previous genuine global total order multicast algorithms that we know of, either do not tolerate failures (e.g [21]), or assume reliable failure detectors (e.g. [11, 15]).

MTO is based on a reduction to the consensus problem. This means that, given *TO-multicast*$(m, \mathsf{Dst}(m))$ and using the results established by Chandra-Toueg [5], MTO requires only (1) (unreliable) failure detectors of the class $\diamond\mathcal{S}$, and (2) within $\mathsf{Dst}(m)$, a qualified majority of correct processes. Interestingly, the correctness of the MTO algorithm requires causal order delivery of messages. This does not mean that any total order genuine multicast algorithm, based on a failure detector of class $\diamond\mathcal{S}$, requires causal order delivery. We suspect, however, that causal order delivery is necessary, but this has still to be formally proved, and constitutes an interesting open question.

The cost of the MTO algorithm (number of communication steps and number of messages) is indeed higher than the cost of Chandra-Toueg's total order broadcast algorithm. However, the fact that MTO is a "genuine" multicast algorithm establishes an interesting trade-off between total order "genuine multicast" and total order "broadcast" algorithms. Finally, whereas many published total order algorithms are based on a membership service, neither Chandra-Toueg's algorithm, nor MTO, require a membership service. Establishing exactly the role of a membership service for total order algorithms is another interesting open question.

References

[1] Y. Amir, L.E. Moser, P.M. Melliar-Smith, D.A. Agarwal, and P.Ciarfella. The Totem Single-Ring Ordering and Membership Protocol. *ACM Trans. on Computer Systems*, 13(4):311–342, November 1995.

[2] K. Birman and T. Joseph. Reliable Communication in the Presence of Failures. *ACM Trans. on Computer Systems*, 5(1):47–76, February 1987.

[3] K. Birman, A. Schiper, and P. Stephenson. Lightweight Causal and Atomic Group Multicast. *ACM Trans. on Computer Systems*, 9(3):272–314, August 1991.

[4] T. D. Chandra, V. Hadzilacos, and S. Toueg. The Weakest Failure Detector for Solving Consensus. In *Proc. 11th annual ACM Symposium on Principles of Distributed Computing*, pages 147–158, 1992.

[5] T.D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. Technical Report 95-1535, Department of Computer Science, Cornell University, August 1995. A preliminary version appeared in the *Proceedings of the Tenth ACM Symposium on Principles of Distributed Computing*, pages 325–340. ACM Press, August 1991.

[6] K. M. Chandy and J. Misra. How processes learn. *Distributed Computing*, 1(1):40–52, 1986.

[7] J. M. Chang and N. Maxemchuck. Reliable Broadcast Protocols. *ACM Trans. on Computer Systems*, 2(3):251–273, August 1984.

[8] F. Cristian, H. Aghili, R. Strong, and D. Dolev. Atomic broadcast: From simple message diffusion to byzantine agreement. In *IEEE 15th Int Symp on Fault-Tolerant Computing (FTCS-15)*, pages 200–206, June 1985.

[9] D. Dolev, S. Kramer, and D. Malki. Early Delivery Totally Ordered Broadcast in Asynchronous Environments. In *IEEE 23rd Int Symp on Fault-Tolerant Computing (FTCS-23)*, pages 544–553, June 1993.

[10] P. Ezhilchelvan, R. Macedo, and S. Shrivastava. Newtop: A Fault-Tolerant Group Communication Protocol. In *IEEE 15th Intl. Conf. Distributed Computing Systems*, May 1995.

[11] H. Garcia-Molina and A. Spauster. Ordered and Reliable Multicast Communication. *ACM Trans. on Computer Systems*, 9(3):242–271, August 1991.

[12] A. S. Gopal. *Fault-Tolerant Broadcast and Multicasts: The Problem of Inconsistency and Contamination.* PhD thesis, Cornell University, Ithaca, NY, March 1992.

[13] R. Guerraoui and A. Schiper. Genuine Total Order Multicast. Technical report, EPFL, Dept d'Informatique, 1996. To appear.

[14] V. Hadzilacos and S. Toueg. Fault-Tolerant Broadcasts and Related Problems. Technical Report 94-1425, Department of Computer Science, Cornell University, May 1994.

[15] X. Jia. A Total Ordering Multicast Protocol Using Propagation Trees. *IEEE Trans. Parallel & Distributed Syst.*, 6(6):617–627, June 95.

[16] M. F. Kaashoek, A. S. Tanenbaum, S. F. Hummel, and H. E. Bal. An Efficient Reliable Broadcast Protocol. *Operating Systems Review*, 23(4):5–19, October 1989.

[17] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Comm. ACM*, 21(7):558–565, July 78.

[18] S. W. Luan and V. D. Gligor. A Fault-Tolerant Protocol for Atomic Broadcast. *IEEE Trans. Parallel & Distributed Syst.*, 1(3):271–285, July 90.

[19] A. Schiper, J. Eggli, and A. Sandoz. A new algorithm to implement causal ordering. In *3rd Intl. Workshop on Distributed Algorithms (WDAG-3)*, pages 219–232. Springer Verlag, LNCS 392, September 1989.

[20] A. Schiper and M. Raynal. From Group Communication to Transactions in Distributed Systems. *Comm. ACM*, April 1996. To appear.

[21] D. Skeen. Unpublished communication, Feb. 1985.

[22] P. Stephenson. *Fast Ordered Multicasts.* PhD thesis, Cornell University, Ithaca, NY, May 1991.

[23] A. S. Tanenbaum. *Distributed Operating Systems.* Prentice Hall, 1995.

[24] U. Wilhelm and A. Schiper. A Hierarchy of Totally Ordered Multicasts. In *14th IEEE Symp. on Reliable Distributed Systems (SRDS-14)*, pages 106–115, Bad Neuenahr, Germany, September 1995.

A Appendix: Proof of the MTO algorithm

Lemma 4.1 *Consider the execution of* TO-multicast$(m, \mathsf{Dst}(m))$*. If* $p_j \in \mathsf{Dst}(m)$ *is correct and R-delivers* m*, then every correct process in* $\mathsf{Dst}(m)$ *eventually decides* $sn(m)$*.*

Proof. By the property of reliable multicast, every correct process $p_j \in \mathsf{Dst}(m)$ eventually R-delivers m, and calls the MTO-SN function (Fig. 3). By line 2 of the MTO-SN function, every correct process sends the ts_j time-stamp to $\mathsf{Dst}(m)$. As we assume, among $\mathsf{Dst}(m)$, a qualified majority of correct processes, every correct process in $\mathsf{Dst}(m)$ eventually gets the ts_k time-stamp from a qualified majority of $\mathsf{Dst}(m)$ (line 3). Thus every correct process in $\mathsf{Dst}(m)$ eventually calls the QM-unif-consensus function (line 5). As we assume a qualified majority of correct processes and a failure detector of class $\diamond\mathcal{S}$, the QM-unif-consensus algorithm eventually terminates [5], and hence every correct process eventually decides $sn(m)$. $\square$

Lemma 4.2 *Consider the execution of* TO-multicast$(m, \mathsf{Dst}(m))$ *by* p_i*. If* p_i *is correct, then every correct process in* $\mathsf{Dst}(m)$ *eventually decides* $sn(m)$*.*

PROOF. Follows directly from Lemma 4.1 and from the property of reliable multicast. $\square$

Proposition 4.3 (TO2) *The MTO algorithm satisfies the Uniform Agreement condition TO2.*

PROOF. Consider *TO-multicast*$(m, \mathsf{Dst}(m))$ and a process $p_k \in \mathsf{Dst}(m)$ that has TO-delivered m. Thus p_k has delivered $sn(m)$, which means that a qualified majority of $\mathsf{Dst}(m)$ has sent the time-stamp $ts(\textit{R-deliver}(m))$: so a qualified majority of $\mathsf{Dst}(m)$ has R-delivered m. As we assume, among $\mathsf{Dst}(m)$, a qualified majority of correct processes, at least one correct process in $\mathsf{Dst}(m)$ has R-delivered m. By Lemma 4.1, every correct process p_j eventually decides $sn(m)$.

We prove, by induction on $x \stackrel{\text{def}}{=} sn(m)$, that m is eventually TO-delivered by p_j.

i) Base step (x=1). As there exists no message m' with $sn(m') < sn(m)$, then m can be TO-delivered by p_j as soon as $sn(m)$ is delivered.

ii) Induction step. Consider $sn(m) = x + 1$. By the induction hypothesis every message m', such that $sn(m') \leq x$, is eventually TO-delivered by p_j. Once all these messages m' have been TO-delivered, then m can be TO-delivered by p_j. $\square$

Proposition 4.4 (TO3) *The MTO algorithm satisfies the Validity condition TO3.*

PROOF. Similar to the proof of Proposition 4.3. $\square$

Lemma 4.5 *Consider the two primitives TO-multicast*$(m_1, \mathsf{Dst}(m_1))$ *and TO-multicast*$(m_2, \mathsf{Dst}(m_2))$*, and assume* $sn(m_1) < sn(m_2)$*. If* p_i *is a process in* $\mathsf{Dst}(m_1) \cap \mathsf{Dst}(m_2)$*, then* p_i R-delivers m_1 *before* TO-delivering m_2.

PROOF. The proof is based on the messages m_1, m_2, the events e_x, e'_x, and the processes p_i, p_x and p_j that appear in Figure 4.

i) Consider the message m_1. Let $p_j \in \mathsf{Dst}(m_1)$ be such that the decision value $sn(m_1)$, computed by the function MTO-SN, is the value $prop_j$ of process p_j (Fig. 3, line 4). At line 3, process p_j waits for the time-stamp ts_k from a qualified majority of $\mathsf{Dst}(m_1)$, before defining $prop_j$. Let $QM(m_1)$ be this qualified majority.

ii) Consider the message m_2, and $p_i \in \mathsf{Dst}(m_1) \cap \mathsf{Dst}(m_2)$. Assume that p_i TO-delivers m_2 at local time t. Thus $sn(m_2)$ is decided by p_i at time t, and by the property of the QM-unif-consensus algorithm, a qualified majority of $\mathsf{Dst}(m_2)$ knows the value $sn(m_2)$ at time t. Let

$QM(m_2)$ be this qualified majority.

iii) As $\mathsf{Dst}(m_1) \cap \mathsf{Dst}(m_2) \neq \emptyset$, we have $QM(m_1) \cap QM(m_2) \neq \emptyset$. Let p_x be a process in $QM(m_1) \cap QM(m_2)$.

By the definition of $QM(m_1)$, p_j delivers the messages carrying the time-stamp $ts(deliver_x(m_1))$ from the processes in $QM(m_1)$ (i.e. from $p_x \in QM(m_1)$) before defining $prop_j$. Let e_x be the event by which p_x R-delivers the message m_1 (see Fig. 4). We note (v) the message carrying the value v. By the definition of $QM(m_2)$, we have $deliver_x((sn(m_2))) \rightarrow decide_i(sn(m_2))$ (see Fig. 4) [9]. Let e'_x be the event by which p_x delivers $(sn(m_2))$.

Figure 4 shows the two possible scenarios: either event e_x happens before e'_x (scenario 1), or e'_x happens before e_x (scenario 2). In scenario 2, process p_x delivers a message carrying $sn(m_2)$ before R-delivering the message m_1. Thus $ts(R\text{-}deliver_x(m_1)) > sn(m_2)$. Since $sn(m_1)$ is computed as the maximum of the delivery time-stamps, we have $sn(m_1) \geq sn(m_2)$, in contradiction with the assumption $sn(m_1) < sn(m_2)$ of the lemma. Thus scenario 2 cannot occur.

In scenario 1, we have $R\text{-}deliver_x(m_1) \rightarrow e'_x \rightarrow decide_i(sn(m_2))$. As m_1 is also sent to p_i, we can apply Lemma A.1 about causal order delivery (see below), with $m \equiv m_1$ and $e_i \equiv decide_i(sn(m_2))$, which leads to $R\text{-}deliver_i(m1) \rightarrow decide_i(sn(m_2))$. Thus m_1 is R-delivered at p_i before $sn(m_2)$ is decided, i.e. before m_2 is TO-delivered. □

Proposition 4.6 *The MTO algorithm satisfies the ordering property TO1.*

PROOF. Global total order is ensured if messages are TO-delivered in the order defined by their sequence number. Consider TO-multicast$(m, \mathsf{Dst}(m))$, TO-multicast$(m', \mathsf{Dst}(m'))$, $sn(m') < sn(m)$, and $p_i \in \mathsf{Dst}(m) \cap \mathsf{Dst}(m')$. Lemma 4.5 ensures that p_i R-delivers m' before TO-delivering m. Thus when m is TO-delivered, m' is either (1) in p_i's pending buffer, or (2) in p_i's delivery buffer, or (3) already TO-delivered. As $sn(m') < sn(m)$, (1) and (2) are excluded by the TO-delivery condition of MTO. Thus, only case (3) is possible, i.e. m' has already been TO-delivered when m is TO-delivered. □

Lemma A.1 *Consider a message m R-multicast to* $\mathsf{Dst}(m)$*, $p_i, p_x \in$* $\mathsf{Dst}(m)$*, and $R\text{-}deliv_x(m)$ the R-delivery of m at p_x. Consider further an event e_i on p_i such that $R\text{-}deliv_x(m) \rightarrow e_i$.*

[9] $deliver_x((sn(m_2)))$ is noted $deliver_x(sn(m_2))$ in Fig. 4.

Causal delivery ensures that either e_i is the R-delivery of m, or $R\text{-}deliv_i(m) \rightarrow e_i$.

PROOF. If e_i is the R-delivery of m the result is established. Otherwise, consider any causal chain of events connecting $R\text{-}deliv_x(m)$ to e_i: $[e^0 \equiv R\text{-}deliv_x(m)] \rightarrow e^1 \rightarrow \ldots \rightarrow e^{l-1} \rightarrow [e^l \equiv e_i]$. Consider the latest event e^{l_0} on this causal chain that is not an event on p_i. By definition, e^{l_0} is the sending of a message m' by some process p_k (i.e. $e^{l_0} \equiv send_k(m')$), and $e^{l_0+1} \equiv deliv_i(m')$. Thus we have: (1) $[e^0 \equiv R\text{-}deliv_x(m)] \rightarrow [e^{l_0} \equiv send_k(m')]$, and (2) $[e^{l_0+1} \equiv deliv_i(m')] \rightarrow [e^l \equiv e_i]$.

By definition, $send(m) \rightarrow [e^0 \equiv R\text{-}deliv_x(m)]$. Together with (1), we have by transitivity $send(m) \rightarrow [e^{l_0} \equiv send_k(m')]$, and m, m' both sent to p_i. Causal order delivery ensures that $R\text{-}deliv_i(m) \rightarrow [deliv_i(m') \equiv e^{l_0+1}]$. Together with (2), we get $R\text{-}deliv_i(m) \rightarrow [e^l \equiv e_i]$. $\square$