

## Handling message semantics with Generic Broadcast protocols

F. Pedone<sup>1</sup>, A. Schiper<sup>2</sup>

<sup>1</sup> Hewlett-Packard Laboratories, Software Technology Laboratory, Palo Alto, CA 94304, USA (e-mail: pedone@hpl.hp.com)

<sup>2</sup> Communication Systems Department, EPFL – Ecole Polytechnique Fédérale de Lausanne, 1015 Lausanne, Switzerland (e-mail: andre.schiper@epfl.ch)

Received: August 2000 / Accepted: August 2001

**Summary.** Message ordering is a fundamental abstraction in distributed systems. However, ordering guarantees are usually purely “syntactic,” that is, message “semantics” is not taken into consideration despite the fact that in several cases semantic information about messages could be exploited to avoid ordering messages unnecessarily. In this paper we define the *Generic Broadcast* problem, which orders messages only if needed, based on the semantics of the messages. The semantic information about messages is introduced by conflict relations. We show that Reliable Broadcast and Atomic Broadcast are special instances of Generic Broadcast. The paper also presents two algorithms that solve Generic Broadcast.

**Keywords:** Semantics-aware primitives – Group communication – Fault-tolerance – Atomic broadcast – Reliable broadcast – Asynchronous systems

### 1 Introduction

Message ordering is a fundamental abstraction in distributed systems. Total order, causal order, and view synchrony are examples of widely used ordering guarantees. These ordering guarantees, however, rely only on “syntactic” information about the messages, ignoring their “semantics.” In general, ordering messages without taking their semantics into consideration leads to ordering more messages than actually necessary to ensure the correctness of the application. Moreover, as ordering messages has a cost, ordering messages unnecessarily penalizes the application. Consider for example a replicated object implemented using active replication – also called state machine approach [12]. By distinguishing messages containing *read* operations from messages containing *write* operations, one could design a protocol that does not order all messages, since read operations do not need to be ordered with respect to other read operations.

A preliminary version of this paper appeared in *Proceedings of the 13th International Symposium on Distributed Computing (DISC’99*, pp. 94–108).

This paper introduces *Generic Broadcast*, a message ordering abstraction that allows applications to specify order requirements based on the semantics of messages. Message ordering requirements are formalized by a *message conflict relation* defined over the set of messages. Roughly speaking, two messages have to be delivered in the same order only if they conflict. The definition of message ordering based on a conflict relation allows for a very powerful message ordering abstraction. For example, the Reliable Broadcast problem is an instance of Generic Broadcast in which no pair of messages conflict. The Atomic Broadcast problem is another instance of Generic Broadcast in which all pairs of messages conflict.

The interest in taking application semantics into account to define more flexible message ordering primitives in group communication was first pointed out in [5]. In [8], the authors consider the issue of ordering messages from the viewpoint of database concurrency control. The notion of message conflict is introduced to capture application semantics, and is used to extend the definitions of FIFO, causal, and total order message delivery to include message semantics. Serialization graphs are used to reason about application correctness along the same lines of database concurrency control [2]. The authors also briefly discuss how one could implement such specifications in a distributed system where processes do not fail. Contrary to [8], we consider here a system model with process failures.

Any algorithm that solves Atomic Broadcast trivially solves any instance of Generic Broadcast (i.e., specified by a given conflict relation): it just orders too many messages. However, such an algorithm goes against the main motivation of Generic Broadcast, which is to allow for efficient message delivery by not ordering messages unnecessarily. We present two algorithms that solve Generic Broadcast, called  $\mathcal{GB}$  and  $\mathcal{GB}+$ ; both algorithms are more efficient than Atomic Broadcast when messages do not conflict.  $\mathcal{GB}$  and  $\mathcal{GB}+$  rely on Consensus [4] when conflicts are detected, but can deliver non-conflicting messages without using Consensus.  $\mathcal{GB}+$  improves the performance of  $\mathcal{GB}$  by being able, in some cases, to deliver conflicting messages without Consensus. This last result is very interesting, as it exhibits an algorithm that can sometimes solve Atomic Broadcast (an instance of Generic Broadcast) in an asynchronous system with process crashes.

Our Generic Broadcast algorithms require  $f < n/3$ , where  $n$  is the total number of processes and  $f$  the maximum number of faulty processes. If messages do not conflict, the algorithms  $\mathcal{GB}$  and  $\mathcal{GB}^+$  have a time complexity of  $2\delta$ , where  $\delta$  is the maximum network message delay [1]. In case of conflicts, the time complexity is  $4\delta$  in the best case, and  $7\delta$  in the worst case. These results are to be compared with the time complexity of Atomic Broadcast algorithms in the model we consider:  $3\delta$  in the best case and  $5\delta$  in the worst case. These results, which show the advantage of Generic Broadcast over Atomic Broadcast if message conflicts are not too frequent, have been validated by a small prototypical implementation.

The work in [1] is close to the one presented in this paper: actually, [1] builds upon [10], the preliminary version of this paper. The work presented in [1] uses an Atomic Broadcast oracle (instead of Consensus, as we do) as the building block for Generic Broadcast, and formalizes classes of Generic Broadcast algorithms according to how they use this oracle. Informally, an algorithm solving Generic Broadcast is *non-trivial* w.r.t. an oracle if, when no conflicting messages are g-Broadcast, the oracle is not used; an algorithm is *thrifty* w.r.t. an oracle if it is non-trivial w.r.t. the oracle and guarantees the following property: if there is a time after which messages g-Broadcast do not conflict with each other, then eventually the oracle is no longer used. Non-trivial and thrifty implementations of Generic Broadcast are given in [1]. The two Generic Broadcast algorithms given in this paper are also thrifty implementations of Generic Broadcast, if we extend the oracle in the definitions of [1] to include a Consensus oracle. From the point of view of time complexity, [1] does not improve our results. The best algorithm in [1] has a time complexity of  $2\delta$  and also requires  $f < n/3$ ; [1] also gives an algorithm for Generic Broadcast with  $f < n/2$ , which has a time complexity of  $3\delta$ .

The remainder of the paper is structured as follows. Section 2 describes the system model and defines the Generic Broadcast problem. Sections 3 and 4 present the two Generic Broadcast algorithms  $\mathcal{GB}$  and  $\mathcal{GB}^+$ , and Sect. 5 contains their proofs of correctness. Section 6 evaluates the time complexity of the two algorithms, and points out the cost of  $\mathcal{GB}$  and  $\mathcal{GB}^+$  with respect to Atomic Broadcast algorithms. Section 7 concludes the paper.

## 2 System model and definitions

### 2.1 Model assumptions

We consider an asynchronous system composed of  $n$  processes  $\Pi = \{p_1, \dots, p_n\}$ , which communicate by message passing. A process can only fail by crashing (i.e., we do not consider Byzantine failures). A process that never crashes is *correct*, otherwise it is *faulty*. We make no assumptions about process speeds or message transmission times.

Processes are connected through quasi-reliable channels, defined by the primitives  $send(m)$  and  $receive(m)$ . Messages are unique and taken from a set  $\mathcal{M}$ . Quasi-reliable channels have the following properties: (i) if process  $q$  receives message  $m$  from  $p$ , then  $p$  sent  $m$  to  $q$  (*no creation*); (ii)  $q$  receives  $m$  from  $p$  at most once (*no duplication*); and (iii) if  $p$  sends  $m$  to  $q$ , and  $p$  and  $q$  are correct, then  $q$  eventually receives  $m$  (*no loss*).

We assume that our asynchronous system is augmented with further abstractions (e.g., failure detectors) allowing us to solve Uniform Consensus [4]. Uniform Consensus is defined by the primitives  $propose(v)$  and  $decide(v)$ , and the following properties: (i) every correct process eventually decides some value (*termination*); (ii) every correct process decides at most once (*uniform integrity*); (iii) no two processes decide differently (*uniform agreement*); and (iv) if a process decides  $v$ , then  $v$  was proposed by some process (*uniform validity*).

### 2.2 Generic Broadcast

Generic Broadcast is defined by the two primitives  $g\text{-Broadcast}(m)$  and  $g\text{-Deliver}(m)$ .<sup>1</sup> When a process  $p$  invokes  $g\text{-Broadcast}$  with a message  $m$ , we say that  $p$   $g\text{-Broadcasts}$   $m$ , and when  $p$  returns from the execution of  $g\text{-Deliver}$  with message  $m$ , we say that  $p$   $g\text{-Delivers}$   $m$ . Message  $m$  is taken from a set  $\mathcal{M}$  to which all messages belong. Generic Broadcast depends on a (symmetric and non-reflexive) conflict relation on  $\mathcal{M} \times \mathcal{M}$  denoted by  $\sim$  (i.e.,  $\sim \subseteq \mathcal{M} \times \mathcal{M}$ ).<sup>2</sup> If  $(m, m') \in \sim$  then we say that  $m$  and  $m'$  conflict. To simplify, we use hereafter the infix notation  $m \sim m'$  instead of  $(m, m') \in \sim$ . Generic Broadcast is specified by (1) a conflict relation  $\sim$  and (2) the following conditions:

- (VALIDITY) If a correct process  $p$   $g\text{-Broadcasts}$  a message  $m$ , then  $p$  eventually  $g\text{-Delivers}$   $m$ .
- (UNIFORM AGREEMENT) If a process  $p$   $g\text{-Delivers}$  a message  $m$ , then every correct process  $q$  eventually  $g\text{-Delivers}$   $m$ .
- (UNIFORM INTEGRITY) For any message  $m$ , every process  $g\text{-Delivers}$   $m$  at most once, and only if  $m$  was previously  $g\text{-Broadcast}$  by some process.
- (UNIFORM ORDER) If processes  $p$  and  $q$  both  $g\text{-Deliver}$  conflicting messages  $m$  and  $m'$ , then  $p$  and  $q$   $g\text{-Deliver}$   $m$  and  $m'$  in the same order.

The conflict relation  $\sim$  determines the pair of messages that are sensitive to order, that is, the pair of messages for which the  $g\text{-Deliver}$  order should be the same at all processes that  $g\text{-Deliver}$  the messages. The conflict relation  $\sim$  renders the above specification generic, as shown next.

### 2.3 Reliable and Atomic Broadcast as instances of Generic Broadcast

We consider in the following two special cases of conflict relations: (1) the empty conflict relation, denoted by  $\sim_\emptyset$  (i.e.,  $\sim_\emptyset \equiv \emptyset$ ), and (2) the cross product conflict relation, denoted by  $\sim_{\mathcal{M} \times \mathcal{M}}$  (i.e.,  $\sim_{\mathcal{M} \times \mathcal{M}} \equiv \mathcal{M} \times \mathcal{M}$ ). In case (1), no pair of messages conflict, that is, the uniform order property of Generic Broadcast imposes no constraints on the order of messages, which is called *Reliable Broadcast* [7] – or, more precisely, *Uniform Reliable Broadcast*. In case (2), any pair  $(m, m')$  of messages conflict, that is, the uniform order property of

<sup>1</sup>  $g\text{-Broadcast}$  has no relation with the GBCAST primitive defined in the Isis system [3].

<sup>2</sup> The operand  $\sim$  was introduced in [1].

Generic Broadcast requires that all pairs of messages be ordered, which is called *Atomic Broadcast* [7] – or, *Uniform Atomic Broadcast*. In other words, Reliable Broadcast and Atomic Broadcast lie at the two ends of the spectrum defined by Generic Broadcast. In between, any other conflict relation defines an instance of Generic Broadcast.

Conflict relations lying in between the two extremes of the conflict spectrum can be better illustrated by an example. Consider a replicated *Account* object, defined by the operations *deposit*( $x$ ) and *withdraw*( $x$ ). Clearly, *deposit* operations commute with each other, while *withdraw* operations do not – neither with each other nor with *deposit* operations.<sup>3</sup> Let  $\mathcal{M}_d$  denote the set of messages that carry a *deposit* operation, and  $\mathcal{M}_w$  the set of messages that carry a *withdraw* operation. This leads to the following conflict relation  $\sim_{Account}$ :

$$\sim_{Account} = \{ (m, m') : m \in \mathcal{M}_w \text{ or } m' \in \mathcal{M}_w \}.$$

Generic Broadcast with the  $\sim_{Account}$  conflict relation defines a weaker ordering primitive than Atomic Broadcast (e.g., messages in  $\mathcal{M}_d$  are not required to be ordered with respect to each other), and a stronger ordering primitive than Reliable Broadcast (e.g., messages in  $\mathcal{M}_w$  have to be ordered with each other).

### 3 $\mathcal{GB}$ : a Generic Broadcast algorithm

In this section and in the next one, we present two Generic Broadcast algorithms:  $\mathcal{GB}$  and  $\mathcal{GB}+$ , respectively. Both algorithms are parameterized by two constants,  $n_{ack}$  and  $n_{chk}$ . From the relationship between  $n_{ack}$  and  $n_{chk}$  – explained later – both algorithms require at least  $(2n + 1)/3$  correct processes, which corresponds to the case where  $n_{ack} = n_{chk} = \lceil (2n + 1)/3 \rceil$ .

#### 3.1 Overview of the $\mathcal{GB}$ algorithm

We start by illustrating the  $\mathcal{GB}$  algorithm with a run in which only two messages are g-Broadcast, and then generalize for the case of  $n$  messages. The algorithm uses Reliable Broadcast, defined by the primitives R-broadcast and R-deliver [7].<sup>4</sup>

*Run with 2 messages.* Consider a run in which only messages  $m$  and  $m'$  are g-Broadcast. The g-Broadcast of message  $m$  leads to the execution of R-broadcast( $m$ ). Upon R-delivery of  $m$  by some process  $p_i$ , there are three cases to consider:

1.  $p_i$  has not R-delivered message  $m'$ ,
2.  $p_i$  has R-delivered message  $m'$ , and  $m'$  does not conflict with  $m$ , or
3.  $p_i$  has R-delivered message  $m'$ , and  $m'$  conflicts with  $m$ .

In cases 1 and 2,  $p_i$  sends a message to all processes acknowledging the R-delivery of  $m$  – hereafter such a message is denoted  $ACK(m)$ . A process that receives  $ACK(m)$  from  $n_{ack}$

<sup>3</sup> This is the case for instance if we consider that a *withdraw*( $x$ ) operation can only be performed if the current balance is larger than or equal to  $x$ .

<sup>4</sup> Reliable Broadcast satisfies the validity, agreement (if a process R-delivers a message  $m$ , then every correct process eventually R-delivers  $m$ ) and uniform integrity properties (Sect. 2).

processes g-Delivers  $m$ . In a run in which no process falls into case 3 above, all correct processes eventually receive  $n_{ack}$  messages  $ACK(m)$  and g-Deliver  $m$ .

In case 3,  $p_i$  launches an instance of Consensus to decide on the g-Delivery order of  $m$  and  $m'$ . This should be done carefully because if some process has already g-Delivered  $m'$ , then  $p_i$  should g-Deliver  $m'$  before  $m$ . Thus, before executing Consensus, every process  $p_i$  sends to all processes a message – hereafter denoted  $CHK$ , containing all messages  $m$  such that  $ACK(m)$  was sent by  $p_i$ . Process  $p_i$  then waits for  $CHK$  messages from  $n_{chk}$  processes.

Upon receiving  $n_{chk}$  messages  $CHK$ , process  $p_i$  builds a set of messages, denoted by  $msgSet_i$ . Set  $msgSet_i$  contains message  $m$  if  $m$  is in a majority of the  $n_{chk}$  messages of type  $CHK$  received by  $p_i$ . As shown next, this ensures that if some process has g-Delivered  $m$ ,  $m \in msgSet_i$ .

To understand  $msgSet_i$ , consider  $n = 4$ ,  $n_{ack} = n_{chk} = 3$ , and assume that process  $p_j$  has g-Delivered  $m$ . So  $p_j$  has received  $n_{ack}$  messages  $ACK(m)$ , i.e., 3 processes have sent  $ACK(m)$ . So, if  $p_i$  waits for  $n_{chk}$  messages of type  $CHK$ , it will get at least 2 messages containing  $m$ . So  $p_i$  includes  $m$  in  $msgSet_i$ .

After building set  $msgSet_i$ ,  $p_i$  executes Consensus proposing  $(msgSet_i, conflictSet_i)$ , where  $conflictSet_i$  contains all messages that  $p_i$  R-delivered and are not in  $msgSet_i$  – that is,  $conflictSet_i = \{m'\}$ . Let  $(NCset, Cset)$  be the Consensus' decision –  $NC$  stands for Non-Conflicting, as this set never contains conflicting messages, and  $C$  stands for Conflicting. Process  $p_i$  g-Delivers first the messages in  $NCset$ , it has not yet g-Delivered, and then the messages in  $Cset$ .

*Generalizing for  $n$  messages.* A run of algorithm  $\mathcal{GB}$  is decomposed into a sequence of two *phases*: the first phase – phase I – lasts as long as no conflicting messages are R-delivered; the second phase – phase II – handles the g-Delivery of conflicting messages thanks to the execution of a Consensus algorithm. These two phases define a *stage*. So, processes progress in a sequence of stages, numbered  $1, \dots, k, \dots$ . In the run considered in the previous paragraph (2 messages), we have one single stage. When some process  $p_i$  starts stage  $k$ , it is initially in phase I. Phase I terminates at process  $p_i$  iff  $p_i$  R-delivers two conflicting messages. In phase II of stage  $k$ , process  $p_i$  first builds  $msgSet_i$  and  $conflictSet_i$ , as described in the previous paragraph, and then executes a Consensus with  $(msgSet_i, conflictSet_i)$  as the initial value. When Consensus terminates with a decision  $(NCset, Cset)$ ,  $p_i$  g-Delivers messages in  $NCset$  not yet g-Delivered, then those in  $Cset$  not yet g-Delivered, and proceeds to phase I of stage  $k + 1$ .

*The parameters  $n_{ack}$  and  $n_{chk}$ .* The  $\mathcal{GB}$  algorithm requires (1)  $n_{ack} > n/2$ , (2)  $n_{chk} > n/2$ , (3)  $2n_{ack} + n_{chk} \geq 2n + 1$ , and (4)  $\max(n_{ack}, n_{chk})$  correct processes. Condition (1) guarantees that if  $m$  and  $m'$  conflict, at most one of them can be g-Delivered without Consensus. Condition (2) ensures that  $msgSet_i$ , constructed by  $p_i$  before Consensus, does not contain conflicting messages. Condition (3) ensures that if some process, say  $p_j$ , has g-Delivered  $m$  before Consensus, and  $m$  conflicts with  $m'$ , then for every process  $p_i$  we have  $m \in msgSet_i$ . Thus, after Consensus, every process first g-Delivers  $m$ . Condition (4) ensures that no wait statement in the algorithm lasts forever. The minimum of condition (4) is for

$n_{ack} = n_{chk}$ . From this and (3), we get that that our algorithm requires at least  $\lceil (2n + 1)/3 \rceil$  correct processes.

### 3.2 The $\mathcal{GB}$ algorithm in detail

We present now the  $\mathcal{GB}$  algorithm (see Fig. 1). Messages are g-Broadcast at line 7 and g-Delivered at lines 22, 23, and 31. The algorithm consists of three concurrent tasks. Process  $p_i$  in stage  $k$  manages the following sets of messages:

- *R.delivered*: set of messages R-delivered by  $p_i$  up to the current time,
- *G.delivered*: set of messages g-Delivered by  $p_i$  in all stages  $k' < k$ ,
- *pending<sup>k</sup>*: set of messages R-delivered by  $p_i$  up to the current time in phase I of stage  $k$  and acknowledged to the other processes, and
- *g.Deliver<sup>k</sup>*: set of messages that  $p_i$  has g-Delivered in phase I of stage  $k$ , up to the current time.

Let process  $p_i$  be in phase I of stage  $k$ . When  $p_i$  wants to g-Broadcast a message  $m$ ,  $p_i$  executes *R-broadcast*( $m$ ) (Fig. 1, line 8). After  $m$  is R-delivered (line 10),  $m$  is included in the sequence *R.delivered* (line 11). Process  $p_i$  then eventually evaluates lines 12 and 13; there are two cases to consider.

*Case 1: no message in  $R.delivered \setminus (G.delivered \cup pending^k)$  conflicts with  $m$ .* In this case,  $p_i$  includes  $m$  in *pending<sup>k</sup>* (line 14), and sends message  $(k, pending^k, ACK)$  to all other processes (line 15), acknowledging that  $m$  does not conflict with any previous message R-delivered by  $p_i$ , but not g-Delivered so far. When a process  $p_j$  receives messages of the type  $(k, pending^k, ACK)$ , with  $m \in pending^k$ , from  $n_{ack}$  processes (lines 28–29),  $p_j$  g-Delivers  $m$ , if it has not done so (line 31).

*Case 2: some message  $m'$  in  $R.delivered \setminus (G.delivered \cup pending^k)$  conflicts with  $m$ .* In this case,  $p_i$  proceeds to phase II (lines 17–27). If one process proceeds to phase II, then the algorithm ensures that all correct processes eventually also proceed to phase II. In phase II, process  $p_i$  sends a message of the type  $(k, pending^k, CHK)$  to all processes (line 17), where *pending<sup>k</sup>* contains all messages that were acknowledged by  $p_i$ , and waits for the receipt of messages of the same type from  $n_{chk}$  processes. Based on the CHK messages received,  $p_i$  determines which messages could have been g-Delivered in phase I by some process (line 19), and executes Consensus (lines 20–21). Messages decided by Consensus and not g-Delivered yet by  $p_i$  are g-Delivered (lines 22–23), and  $p_i$  starts the next stage in phase I (lines 25–27).

## 4 $\mathcal{GB}+$ : Improving the $\mathcal{GB}$ algorithm

We present now  $\mathcal{GB}+$ , an improved version of the  $\mathcal{GB}$  algorithm. To understand the difference between  $\mathcal{GB}$  and  $\mathcal{GB}+$ , consider a run in which only two conflicting messages  $m$  and  $m'$  are g-Broadcast, and  $m$  is g-Delivered by some process  $p_i$

in phase I of stage 1. Assume that later in phase I of stage 1, process  $p_i$  R-delivers  $m'$ . In this case, with  $\mathcal{GB}$ , process  $p_i$  starts phase II to terminate the current stage by an instance of Consensus. However, this is not necessary as the Consensus decision is known beforehand:  $m$  has already been g-Delivered, before  $m'$ . So, while  $p_i$  executing  $\mathcal{GB}$  proceeds to phase II, with  $\mathcal{GB}+$ , process  $p_i$  remains in phase I and may g-Deliver  $m'$  in phase I even though  $m$  and  $m'$  conflict. So,  $\mathcal{GB}+$  can sometimes g-Deliver conflicting messages without Consensus.

### 4.1 The $\mathcal{GB}+$ algorithm

In addition to the sets of messages, *R.delivered*, *G.delivered*, and *pending<sup>k</sup>* of  $\mathcal{GB}$ , the  $\mathcal{GB}+$  algorithm (see Fig. 2) uses also *g.Deliver<sup>k</sup>*, which is a “sequence” of messages. This variable keeps track of the order in which messages are locally g-Delivered at a process. Besides the traditional set operands, we also use the  $\oplus$  operand to append messages to *g.Deliver<sup>k</sup>*.

Tasks 1 and 2 are the same for both  $\mathcal{GB}$  and  $\mathcal{GB}+$ . In Task 3  $\mathcal{GB}$  and  $\mathcal{GB}+$  are similar, except for the following differences:

- Processes executing  $\mathcal{GB}+$  ignore messages that have already been locally g-Delivered in the current stage (lines 13–14) to detect whether Consensus is needed. Moreover, in  $\mathcal{GB}+$ , messages of type ACK have one additional field (*g.Deliver<sup>k</sup>*), to carry the messages that a process has locally g-Delivered so far in the current stage (line 16). This leads to a difference in the *when* clause that treats ACK messages (lines 21–23).
- With  $\mathcal{GB}+$ , it is possible that some process detects a situation where Consensus is needed, and the other processes do not. This happens because the condition to start Consensus depends on the order in which messages are locally R-delivered (which may not be the same for all processes). Thus, a process can start Consensus in two circumstances: either (a) because it detected that Consensus is needed (line 14), or (b) because it received a message of type CHK from some process (line 27), who has detected that Consensus is needed.
- Messages of type CHK (lines 19 and 30) also include an additional field (*g.Deliver<sup>k</sup>*) containing the sequencer of messages g-Delivered so far by the sender in the current stage. Whenever a process receives a message of the type  $(k, g.Deliver^k, pending^k, CHK)$ , it g-Delivers all messages in *g.Deliver<sup>k</sup>* that it has not g-Delivered so far, following the order in *g.Deliver<sup>k</sup>* (lines 31–33). Variable *chk\_flag* is used to make sure that a process only sends a message of type CHK once in a stage.

### 4.2 $\mathcal{GB}+$ as a solution to Atomic Broadcast

By considering an instance of  $\mathcal{GB}+$  where any two messages conflict, we can use  $\mathcal{GB}+$  to solve Atomic Broadcast. Taking into account the properties of  $\mathcal{GB}+$ , we have an Atomic Broadcast algorithm that, in some runs, orders messages without Consensus and without any other assumptions about the model (e.g., failure detectors). Notice that even though this leads to situations where some messages can be ordered in a

```

1: Initialization:

2:  $R\_delivered \leftarrow \emptyset$ 
3:  $G\_delivered \leftarrow \emptyset$ 
4:  $pending^1 \leftarrow \emptyset$ 
5:  $g\_Deliver^1 \leftarrow \emptyset$ 
6:  $k \leftarrow 1$ 

7: To execute  $g$ -Broadcast( $m$ ): {Task 1}

8:  $R$ -broadcast( $m$ )

9:  $g$ -Deliver( $m$ ) occurs as follows:

10: when  $R$ -deliver( $m$ ) {Task 2}
11:    $R\_delivered \leftarrow R\_delivered \cup \{m\}$ 

12: when  $(R\_delivered \setminus (G\_delivered \cup pending^k)) \neq \emptyset$  {Task 3}
13:   if [ for all  $m, m' \in (R\_delivered \setminus G\_delivered) : m \not\sim m'$  ] then
14:      $pending^k \leftarrow R\_delivered \setminus G\_delivered$ 
15:     send( $k, pending^k, ACK$ ) to all
16:   else
17:     send( $k, pending^k, CHK$ ) to all
18:     wait until [ for  $n_{chk}$  processes  $p_j$  : received  $(k, pending_j^k, CHK)$  from  $p_j$  ]
19:      $msgSet^k \leftarrow \{m \mid \text{for } \lceil \frac{n_{chk}+1}{2} \rceil \text{ processes } p_j : \text{received } (k, pending_j^k, CHK) \text{ from } p_j \text{ and } m \in pending_j^k\}$ 
20:     propose( $k, msgSet^k, (R\_delivered \setminus (G\_delivered \cup msgSet^k))$ )
21:     wait until decide( $k, NCset^k, Cset^k$ )
22:     for each  $m \in NCset^k \setminus (g\_Deliver^k \cup G\_delivered)$  do  $g$ -Deliver( $m$ )
23:     in ID order: for each  $m \in Cset^k \setminus (g\_Deliver^k \cup G\_delivered)$  do  $g$ -Deliver( $m$ )
24:      $G\_delivered \leftarrow G\_delivered \cup NCset^k \cup Cset^k$ 

25:    $k \leftarrow k + 1$ 
26:    $pending^k \leftarrow \emptyset$ 
27:    $g\_Deliver^k \leftarrow \emptyset$ 

28: when (receive( $k, pending_j^k, ACK$ ) from  $p_j$ )
29:   while  $\exists m$  such that [ for  $n_{ack}$  processes  $p_j$  : received  $(k, pending_j^k, ACK)$  from  $p_j$  and
 $m \in (pending_j^k \setminus g\_Deliver^k) \mid \text{do}$ 
30:      $g\_Deliver^k \leftarrow g\_Deliver^k \cup \{m\}$ 
31:      $g$ -Deliver( $m$ )

```

Fig. 1. Generic Broadcast algorithm ( $\mathcal{GB}$ )

pure asynchronous model, it is not in contradiction with the FLP impossibility result [6], and the fact that Atomic Broadcast and Consensus are equivalent [4], since it does not apply to all runs.

## 5 Proof of correctness

### 5.1 Proof of correctness of $\mathcal{GB}$

We initially define the following notation, used in Lemmas 1 and 2. Given message  $m$ , we denote by  $ackSet^k(m)$  the set of processes that execute  $send(k, pending^k, ACK)$  (line 17) in stage  $k$ , with  $m \in pending^k$ . Given process  $p_i$ , we denote by  $chkSet^k(p_i)$  the set of processes from which  $p_i$  receives messages of the type  $(k, pending^k, CHK)$  (line 18) in stage  $k$ .

**Lemma 1.** (Assumes  $2n_{ack} + n_{chk} \geq 2n + 1$ .)  
If  $|ackSet^k(m)| = n_{ack}$  and  $|chkSet^k(p_i)| = n_{chk}$ , then there are at least  $\lceil (n_{chk} + 1)/2 \rceil$  processes in the set  $chkSet^k(m, p_i) \stackrel{def}{=} ackSet^k(m) \cap chkSet^k(p_i)$ .

*Proof.* Because  $2n_{ack} + n_{chk} \geq 2n + 1$ , we have  $n_{ack} - n \geq (1 - n_{chk})/2$ . So,  $n_{ack} - n + n_{chk} \geq$

$(1 - n_{chk})/2 + n_{chk} = (n_{chk} + 1)/2$  (a). By definition,  $|chkSet^k(m, p_i)| = |ackSet^k(m) \cap chkSet^k(p_i)|$  and  $|ackSet^k(m) \cap chkSet^k(p_i)| = |ackSet^k(m)| + |chkSet^k(p_i)| - |ackSet^k(m) \cup chkSet^k(p_i)| \geq n_{ack} + n_{chk} - n$ . So we have  $|chkSet^k(m, p_i)| \geq n_{ack} + n_{chk} - n$  (b). From (a) and (b), we have  $|chkSet^k(m, p_i)| \geq (n_{chk} + 1)/2$ , and since  $|chkSet^k(m, p_i)| \in \mathbb{N}$ , it follows that  $|chkSet^k(m, p_i)| \geq \lceil (n_{chk} + 1)/2 \rceil$ .  $\square$

**Lemma 2.** (Assumes  $2n_{ack} + n_{chk} \geq 2n + 1$ .)

If message  $m$  is  $g$ -Delivered by process  $p_i$  in the first phase of stage  $k$ , and  $(k, NCset^k, Cset^k)$  is the value decided in the  $k$ -th execution of Consensus, then  $m \in NCset^k$ .

*Proof.* (uses Lemma 1) Before  $g$ -Delivering  $m$ ,  $p_i$  received  $n_{ack}$  messages of the type  $(k, pending^k, ACK)$  with  $m \in pending^k$  (line 15). Let  $(k, NCset^k, Cset^k)$  be the decision of Consensus of stage  $k$ . From uniform validity of Consensus, there is some process  $p_j$  that has proposed value  $(k, msgSet^k, -) \equiv (k, NCset^k, -)$  at line 20. Before executing  $propose(k, msgSet^k, -)$ ,  $p_j$  has received  $n_{chk}$  messages of the type  $(k, pending^k, CHK)$  in stage  $k$ . From Lemma 1,  $|ackSet^k(m) \cap chkSet^k(p_j)| \geq \lceil (n_{chk} + 1)/2 \rceil$ , and so,  $p_j$  has included  $m$  in  $msgSet^k$ . Thus,  $m \in NCset^k$ .  $\square$

```

1: Initialization:

2:   $R\_delivered \leftarrow \emptyset$ 
3:   $G\_delivered \leftarrow \emptyset$ 
4:   $pending^1 \leftarrow \emptyset$ 
5:   $g\_Deliver^1 \leftarrow \epsilon$ 
6:   $k \leftarrow 1$ 
7:   $chk\_flag \leftarrow false$ 

8: To execute  $g\text{-Broadcast}(m)$ : {Task 1}

9:   $R\text{-broadcast}(m)$ 

10:  $g\text{-Deliver}(-)$  occurs as follows:

11: when  $R\text{-deliver}(m)$  {Task 2}
12:    $R\_delivered \leftarrow R\_delivered \cup \{m\}$ 

13: when  $(R\_delivered \setminus (G\_delivered \cup g\_Deliver^k \cup pending^k)) \neq \emptyset$  {Task 3}
14:   if [ for all  $m, m' \in (R\_delivered \setminus (G\_delivered \cup g\_Deliver^k)) : m \neq m'$  ] then
15:      $pending^k \leftarrow R\_delivered \setminus (G\_delivered \cup g\_Deliver^k)$ 
16:     send( $k, g\_Deliver^k, pending^k, ACK$ ) to all
17:   else
18:      $chk\_flag \leftarrow true$ 
19:     send( $k, g\_Deliver^k, pending^k, CHK$ ) to all

20: when (receive( $k, g\_Deliver_j^k, pending_j^k, ACK$ ) from  $p_j$ ) and not( $chk\_flag$ )
21:   in sequence order: for each  $m \in g\_Deliver_j^k \setminus (g\_Deliver^k \cup G\_delivered)$  do
22:      $g\_Deliver^k \leftarrow g\_Deliver^k \oplus \langle m \rangle$ 
23:      $g\text{-Deliver}(m)$ 
24:     while  $\exists m$  such that [ for  $n_{ack}$  processes  $p_j$  : received( $k, -, pending_j^k, ACK$ ) from  $p_j$  and
 $m \in (pending_j^k \setminus g\_Deliver^k)$  ] do
25:        $g\_Deliver^k \leftarrow g\_Deliver^k \oplus \langle m \rangle$ 
26:        $g\text{-Deliver}(m)$ 

27: when (receive( $k, g\_Deliver_j^k, pending_j^k, CHK$ ) from  $p_j$ )
28:   if not( $chk\_flag$ ) then
29:      $chk\_flag \leftarrow true$ 
30:     send( $k, g\_Deliver^k, pending^k, CHK$ ) to all

31:   in sequence order: for each  $m \in g\_Deliver_j^k \setminus (g\_Deliver^k \cup G\_delivered)$  do
32:      $g\_Deliver^k \leftarrow g\_Deliver^k \oplus \langle m \rangle$ 
33:      $g\text{-Deliver}(m)$ 

34:   if [ for  $n_{chk}$  processes  $p_j$  : received( $k, g\_Deliver_j^k, pending_j^k, CHK$ ) ] then
35:      $msgSet^k \leftarrow \{m \mid \text{for } \lceil \frac{n_{chk}+1}{2} \rceil \text{ processes } p_j : \text{received}(k, g\_Deliver_j^k, pending_j^k, CHK) \text{ and } m \in pending_j^k\}$ 
36:     propose( $k, msgSet^k, (R\_delivered \setminus (G\_delivered \cup msgSet^k))$ )
37:     wait until decide( $k, NCset^k, Cset^k$ )
38:     for each  $m \in NCset^k \setminus (g\_Deliver^k \cup G\_delivered)$  do  $g\text{-Deliver}(m)$ 
39:     in ID order: for each  $m \in Cset^k \setminus (g\_Deliver^k \cup G\_delivered)$  do  $g\text{-Deliver}(m)$ 
40:      $G\_delivered \leftarrow G\_delivered \cup g\_Deliver^k$ 

41:    $k \leftarrow k + 1$ 
42:    $pending^k \leftarrow \emptyset$ 
43:    $g\_Deliver^k \leftarrow \epsilon$ 
44:    $chk\_flag \leftarrow false$ 

```

**Fig. 2.** Improved Generic Broadcast algorithm ( $GB+$ )

**Lemma 3.** (Assumes  $2n_{ack} + n_{chk} \geq 2n + 1$ .)  
If message  $m$  is  $g\text{-Delivered}$  by some process in stage  $k$ , then every process that terminates stage  $k$  (i.e., executes line 24 in stage  $k$ )  $g\text{-Delivers}$   $m$ .

*Proof.* (uses Lemma 2) Let  $k$  be the smallest stage in which some process, say  $p_i$ ,  $g\text{-Delivers}$   $m$  (at lines 22, 23, or 31), and let  $p_j \neq p_i$  be a process that terminates stage  $k$ . Thus,  $p_j$  executes Consensus in stage  $k$  and  $g\text{-Delivers}$  all messages in  $NCset^k \cup Cset^k$ , where  $(NCset^k, Cset^k)$  is the value decided in Consensus in stage  $k$ . There are two cases to con-

sider: (a) If  $p_i$   $g\text{-Delivers}$   $m$  in the first phase of stage  $k$ , from Lemma 2,  $m \in NCset^k$ . (b) If  $p_i$   $g\text{-Delivers}$   $m$  in the second phase of stage  $k$ , then  $m \in NCset^k \cup Cset^k$ . In both cases,  $p_j$   $g\text{-Delivers}$   $m$ .  $\square$

**Lemma 4.** (Assumes  $2n_{ack} + n_{chk} \geq 2n + 1$ ,  $n_{chk} > n/2$ ,  $n_{chk}$  correct processes.)  
For all stage  $k > 0$ , if some process  $p_i$  terminates stage  $k$  (i.e., executes line 24 in stage  $k$ ), then every correct process also terminates stage  $k$ .

*Proof.* (uses Lemma 3) The proof is by induction; however, as the base step is very similar to the inductive step, we only give the proof of the inductive step. Assume the result holds for  $k$ , and let process  $p_i$  terminate stage  $k + 1$ . So  $p_i$  has terminated stage  $k$ , and by the induction hypothesis every correct process also terminates stage  $k$ .

Before terminating stage  $k + 1$ ,  $p_i$  has received  $n_{chk}$  messages  $(k + 1, pending^{k+1}, CHK)$  (line 18). As there are  $n_{chk}$  correct processes and  $n_{chk} > n/2$ , at least one message  $(k + 1, pending^{k+1}, CHK)$  was sent (line 17) by a correct process, say  $p_j$ . Before executing line 17,  $p_j$  has evaluated the condition of line 13 to *false*, that is,  $p_j$  has R-delivered two conflicting messages  $m$  and  $m'$  that are not in  $G.delivered$ , and so,  $p_j$  has not g-Delivered  $m$  and  $m'$  in some stage  $k' < k + 1$ . By the agreement property of Reliable Broadcast, every correct process  $p_r$  eventually also R-delivers  $m$  and  $m'$ . By Lemma 3, as  $p_j$  has not g-Delivered  $m$  and  $m'$  in some previous stage  $k' < k + 1$ , the same holds for  $p_r$ . So every correct process  $p_r$  eventually also evaluates the condition of line 13 to *false*, and sends at the message  $(k + 1, pending^{k+1}, CHK)$  (line 17). As there are  $n_{chk}$  correct processes, every correct process eventually receives (line 18)  $n_{chk}$  such messages and proceed to line 19. So every correct process eventually start Consensus at line 20. By the termination of Consensus every correct process eventually decides, and terminates stage  $k + 1$  at line 25.  $\square$

**Proposition 1.** (UNIFORM AGREEMENT).

(Assumes  $n_{ack}, n_{chk} > n/2$ ,  $max(n_{ack}, n_{chk})$  correct processes,  $2n_{ack} + n_{chk} \geq 2n + 1$ .)

If a process  $p_i$  g-Delivers a message  $m$ , then every correct process  $p_j$  eventually g-Delivers  $m$ .

*Proof.* (uses Lemmas 2, 3, and 4) Process  $p_i$  g-Delivers messages at lines 22, 23 and 31. If  $p_i$  g-Delivers  $m$  at line 22 ( $m \in NCset^k$ ) or at line 23 ( $m \in Cset^k$ ), then  $p_i$  terminates stage  $k$ , and from Lemma 4,  $p_j$  also terminates stage  $k$ . Before terminating stage  $k$ ,  $p_j$  decides for Consensus at line 21; by uniform agreement of Consensus,  $p_j$  also decides with  $m \in NCset^k$  or  $m \in Cset^k$ , and so, also g-Delivers  $m$  at line 22 or 23.

Thus, assume that  $p_j$  does not execute Consensus at stage  $k$ , that is,  $p_i$  g-Delivers  $m$  in the first phase of stage  $k$  (at line 31). We claim that no process evaluates the condition at line 13 to *false* in stage  $k$ . The proof is immediate from the fact that  $p_i$  does not terminate stage  $k$  and from Lemma 4.

As  $p_i$  g-Delivers  $m$  at line 31,  $p_i$  has received messages  $(k, pending^k, ACK)$  with  $m \in pending^k$  from  $n_{ack}$  processes (line 28). Since there are  $n_{ack}$  correct processes and  $n_{ack} > n/2$ ,  $p_i$  has received  $(k, pending^k, ACK)$  from at least one correct process, say  $p_r$ . At line 15,  $p_r$  has sent  $(k, pending^k, ACK)$  with  $m \in pending^k$ , thus at line 14 we have for  $p_r$   $m \in R.delivered_r \setminus G.delivered_r$ , and  $p_r$  has R-delivered  $m$  at line 10. Since  $p_r$  is correct, by the agreement property of Reliable Broadcast every correct process  $p_s$  eventually also R-delivers  $m$ . By Lemma 3,  $p_s$  has not g-Delivered  $m$  in any stage  $k' < k$ , so every correct process  $p_s$  evaluates the condition of line 12 to *true* and starts Task 3.

From our claim above, no process evaluates the condition of line 13 to *false*, and so, every correct process  $p_s$  sends  $(k, pending^k, ACK)$  to all with  $m \in pending^k$  (line 15). As

there are  $n_{ack}$  correct processes,  $p_j$  receives  $n_{ack}$  messages  $(k, pending^k, ACK)$  with  $m \in pending^k$  (line 28), and  $p_j$  g-Delivers  $m$  at line 31.  $\square$

**Lemma 5.** (Assumes  $n_{ack} > n/2$ .)

Let  $m$  and  $m'$  be two conflicting messages, and  $k$  any stage. If process  $p_i$  g-Delivers message  $m$  in the first phase of stage  $k$ , then no process g-Delivers  $m'$  in the first phase of stage  $k$ .

*Proof.* For a contradiction, assume that in the first phase of stage  $k$  process  $p_i$  g-Delivers  $m$  and  $p_j$  g-Delivers  $m'$ . So,  $p_i$  (resp.,  $p_j$ ) has received (line 28)  $n_{ack}$  messages of the type  $(k, pending^k, ACK)$ , such that  $m \in pending^k$  (resp.,  $m' \in pending^k$ ). By the condition of line 13,  $m$  and  $m'$  cannot be in the same set  $pending^k$ , and so, there must exist  $2n_{ack}$  different processes that have sent  $(k, pending^k, ACK)$  at line 15 – a contradiction since  $n_{ack} > n/2$ .  $\square$

**Lemma 6.** (Assumes  $n_{chk} > n/2$ .)

For any stage  $k > 0$ , the set  $NCset^k$  decided in Consensus in stage  $k$  cannot include two conflicting messages.

*Proof.* Let  $m$  and  $m'$  be two conflicting messages. For a contradiction, assume that we have  $m, m' \in NCset^k$ . By the validity property of Consensus some process  $p_i$  has proposed at line 20  $(k, msgSet^k, -) \equiv (k, NCset^k, -)$  such that  $m, m' \in msgSet^k$ . Thus,  $p_i$  receives at line 18  $\lceil \frac{n_{chk}+1}{2} \rceil$  messages  $(k, pending^k, CHK)$  such that  $pending^k$  includes  $m$ , and  $\lceil \frac{n_{chk}+1}{2} \rceil$  messages  $(k, pending^k, CHK)$  such that  $pending^k$  includes  $m'$ . By the condition of line 13,  $pending^k$  cannot include conflicting messages, and so there must exist  $2n_{chk}$  different processes that have sent  $(k, pending^k, CHK)$  at line 19 – a contradiction since  $n_{chk} > n/2$ .  $\square$

**Proposition 2.** (UNIFORM ORDER).

(Assumes  $n_{ack}, n_{chk} > n/2$ ,  $2n_{ack} + n_{chk} \geq 2n + 1$ .)

If processes  $p_i$  and  $p_j$  both g-Deliver conflicting messages  $m$  and  $m'$ , then  $p_i$  and  $p_j$  g-Deliver  $m$  and  $m'$  in the same order.

*Proof.* (uses Lemmas 2, 3, 5, and 6) Without loss of generality, assume that  $p_i$  g-Delivers  $m$  before  $m'$ . If  $p_i$  g-Delivers  $m$  in stage  $k$  and  $m'$  in stage  $k' > k$ , it follows from Lemma 3 that  $p_j$  also g-Delivers  $m$  in stage  $k$  and  $m'$  in stage  $k' > k$  and the result holds.

So, assume that  $p_i$  g-Delivers  $m$  and  $m'$  in stage  $k$ . From Lemma 5  $m$  and  $m'$  cannot be g-Delivered by  $p_i$  in the first phase of stage  $k$ . So, either (1)  $p_i$  g-Delivers  $m$  in the first phase of stage  $k$  and  $m'$  in the second phase of stage  $k$ , or (2)  $p_i$  g-Delivers  $m$  and  $m'$  in the second phase of stage  $k$ .

In case (1), from Lemma 5,  $p_j$  cannot g-Deliver  $m'$  in the first phase of stage  $k$ . From Lemma 2, when  $p_j$  decides at line 21, we have  $m \in NCset^k$ . By Lemma 6,  $NCset^k$  cannot contain  $m$  and  $m'$ . So  $m' \in Cset^k$ , and  $p_j$  also g-Delivers  $m$  (at line 22) before  $m'$  (at line 23).

In case (2), from Lemma 6,  $m$  and  $m'$  cannot both be in  $NCset^k$  decided by  $p_i$ . Therefore, either (2a)  $m \in NCset^k$ ,  $m' \in Cset^k$  decided by Consensus, or (2b)  $m, m' \in Cset^k$  decided by Consensus. In case (2a),  $p_j$  also g-Delivers  $m \in NCset^k$  (line 22) before  $m' \in Cset^k$  (line 23). In case (2b), because  $p_i$  g-Delivers  $m$  before  $m'$ ,  $m$  has a smallest ID than  $m'$ . So  $p_j$  also g-Delivers  $m$  before  $m'$ .  $\square$

**Proposition 3.** (VALIDITY).

(Assumes  $n_{ack}, n_{chk} > n/2$ ,  $\max(n_{ack}, n_{chk})$  correct processes,  $2n_{ack} + n_{chk} \geq 2n + 1$ .)

If a correct process  $p_i$  g-Broadcasts a message  $m$ , then  $p_i$  eventually g-Delivers  $m$ .

*Proof.* (uses Lemma 4 and Proposition 1) For a contradiction, assume that  $p_i$  g-Broadcasts  $m$  but never g-Delivers it. From Proposition 1, no correct process g-Delivers  $m$ . To g-Broadcast  $m$ ,  $p_i$  R-broadcasts it (line 8), and by validity of Reliable Broadcast,  $p_i$  eventually R-delivers  $m$ . From agreement of Reliable Broadcast, every correct process eventually R-delivers  $m$  (line 10). Since no correct process g-Delivers  $m$ , there is a time  $t$  after which, for every correct process  $p_j$ , we have  $m \in (R.delivered \setminus G.delivered)$ .

Let  $t' > t$  be a time such that at  $t'$  all faulty processes have crashed. Let  $k$  be the highest stage reached by some process, say  $p_j$ , at time  $t'$ . From Lemma 4 all correct processes eventually start stage  $k$ .

Since no correct process g-Delivers  $m$ , no correct process g-Delivers  $m$  in the first phase of stage  $k$ . Therefore, no correct process receives  $n_{ack}$  messages  $(k, pending^k, ACK)$  (line 28) such that  $m \in pending^k$ . Since there are  $n_{ack}$  correct processes, at least one correct process, say  $p_j$ , does not send the message  $(k, pending^k, ACK)$  to all with  $m \in pending^k$  (line 15). So  $p_j$  evaluates the condition at line 12 to *false*, and sends the message  $(k, pending^k, CHK)$  to all (line 17), which is only possible if  $p_j$  has R-delivered a message  $m'$  that conflicts with  $m$ . As  $p_j$  is correct, by the agreement property of Reliable Broadcast, every correct process eventually R-delivers  $m'$ , evaluates the condition at line 13 to *false* and sends message  $(k, pending^k, CHK)$  to all. As there are  $n_{chk}$  correct processes, all correct processes eventually stop waiting at line 18 and execute  $propose(k, s, s')$ , with  $m \in s \cup s'$  (line 19). From Consensus, we have at line 21  $m \in NCset^k \cup Cset^k$ . So all correct processes g-Deliver  $m$  at line 22 or 23 – a contradiction.  $\square$

**Proposition 4.** (UNIFORM INTEGRITY). For any message  $m$ , each process g-Delivers  $m$  at most once, and only if  $m$  was previously g-Broadcast.

*Proof.* Assume that  $m$  is never g-Broadcast. So  $m$  is never R-broadcast, and by the uniform integrity of Reliable Broadcast,  $m$  is never R-delivered (line 10). So  $m$  is not in any  $pending^k$  set, and it follows that  $m$  can never be g-Delivered, either at line 31 or at lines 22, 23.

It is not hard to see, from the delivery condition (lines 22, 23, 29), that  $m$  is not g-Delivered more than once.  $\square$

**Theorem 1.** Assume that there are  $\max(n_{ack}, n_{chk})$  correct processes,  $n_{ack}, n_{chk} > n/2$ , and  $2n_{ack} + n_{chk} \geq 2n + 1$ . The algorithm in Fig. 1 solves Generic Broadcast, or reduces Generic Broadcast to a sequence of Consensus problems.

*Proof.* Immediate from Propositions 1, 2, 3, and 4.  $\square$

## 5.2 Proof of correctness of $\mathcal{GB}+$

Since the algorithm  $\mathcal{GB}+$  is derived from  $\mathcal{GB}$ , some results established for  $\mathcal{GB}$  hold for  $\mathcal{GB}+$ :

- Lemmas 1, 2 and 6 hold for  $\mathcal{GB}+$  with the same proof.
- Lemma 3 holds for  $\mathcal{GB}+$ , but the proof requires a tiny adaptation. Indeed, with  $\mathcal{GB}+$ , messages can additionally be g-Delivered at lines 23 and 33. However, this does not require changes in the arguments of the proof of Lemma 3.

Lemma 4 also holds for  $\mathcal{GB}+$ , but the proof is not the same.

**Lemma 4.** (Assumes  $2n_{ack} + n_{chk} \geq 2n + 1$ ,  $n_{chk} > n/2$ ,  $n_{chk}$  correct processes.)

For all stage  $k > 0$ , if some process  $p_i$  terminates stage  $k$  (i.e., executes line 40 in stage  $k$ ), then every correct process also terminates stage  $k$ .

*Proof.* (uses Lemma 3) The proof is by induction; however, as the base step is very similar to the inductive step, we only give the proof of the inductive step. Assume the result holds for  $k$ , and let process  $p_i$  terminate stage  $k + 1$ . So  $p_i$  has terminated stage  $k$ , and by the induction hypothesis every correct process also terminates stage  $k$ . Before terminating stage  $k + 1$ ,  $p_i$  has received  $n_{chk}$  messages  $(k + 1, g\_Deliver^{k+1}, pending^{k+1}, CHK)$  (line 34).

As there are  $n_{chk}$  correct processes and  $n_{chk} > n/2$ , at least one message  $(k + 1, g\_Deliver^{k+1}, pending^{k+1}, CHK)$  was sent (line 19 or 30) by a correct process, say  $p_j$ . So every correct process eventually receives message  $(k + 1, g\_Deliver^{k+1}, pending^{k+1}, CHK)$  (line 27) and sends message  $(k + 1, g\_Deliver^{k+1}, pending^{k+1}, CHK)$  to all (line 30). As there are  $n_{chk}$  correct processes, every correct process eventually evaluates the condition of line 34 to *true* and starts Consensus at line 36. By the termination of Consensus every correct process eventually decides, and terminates stage  $k + 1$  at line 40.  $\square$

**Proposition 5.** (UNIFORM AGREEMENT).

(Assumes  $n_{ack}, n_{chk} > n/2$ ,  $\max(n_{ack}, n_{chk})$  correct processes,  $2n_{ack} + n_{chk} \geq 2n + 1$ .)

If a process  $p_i$  g-Delivers a message  $m$ , then every correct process  $p_j$  eventually g-Delivers  $m$ .

*Proof.* (uses Lemmas 2, 3, and 4) Process  $p_i$  g-Delivers messages at lines 23, 26, 33, 38, and 39. If  $p_i$  g-Delivers  $m$  at line 38 or 39 after the execution of Consensus, the result holds with the same arguments as those for  $\mathcal{GB}$  (Proposition 1). Thus, assume that  $p_i$  does not execute Consensus at stage  $k$ , that is,  $p_i$  g-Delivers  $m$  at line 23, 26, or 33. If  $p_i$  g-Delivers  $m$  at line 23 or 33, there exists a process, say  $p_j$ , that has g-Delivered  $m$  at line 26 after having received  $n_{ack}$  messages of the type  $(k, -, pending^k, ACK)$ . From here on, we can establish the result by using the arguments of the proof of Proposition 1 for the case where  $p_i$  g-Delivers  $m$  at line 31 (Fig. 1) after having received  $n_{ack}$  messages of the type  $(k, -, pending^k, ACK)$ .  $\square$

Lemma 5 no longer holds for  $\mathcal{GB}+$ , as  $\mathcal{GB}+$  allows the g-Delivery of conflicting messages in the first phase of a stage. We replace Lemma 5 with the following lemma:

**Lemma 7.** (Assumes  $n_{ack} > n/2$ .)

Let  $m$  and  $m'$  be two conflicting messages, and  $k$  any stage. If process  $p_i$  g-Delivers  $m$  in stage  $k$  before Consensus, and  $p_j$  g-Delivers  $m'$  in stage  $k$  before Consensus, then either  $p_i$  has g-Delivered  $m'$  before  $m$ , or  $p_j$  has g-Delivered  $m$  before  $m'$ .



*Proof.* For a contradiction assume that  $p_i$  g-Delivers  $m$  before  $m'$ , and  $p_j$  g-Delivers  $m'$  before  $m$ . Processes can g-Deliver messages before Consensus at line 23, 26, or 33.

We first prove that there exists a process  $p_r$  that has g-Delivered  $m$  at line 26, and has not g-Delivered  $m'$  before  $m$ . If  $p_i$  g-Delivers  $m$  at line 26, take  $p_r = p_i$ . If  $p_i$  g-Delivers  $m$  at line 23 or 33, there must exist a process, say  $p_r$ , that has g-Delivered  $m$  at line 26. If  $p_r$  has g-Delivered  $m'$  before  $m$ , then  $p_i$  also g-Delivers  $m'$  before  $m$  at line 23 or 33 (since  $m'$  is before  $m$  in  $g\_Deliver^k$  received from  $p_r$ ) – a contradiction with the fact that  $p_i$  has not g-Delivered  $m'$  before  $m$ . So  $p_r$  has not g-Delivered  $m'$  before  $m$ . By a similar argument, there must exist a process, say  $p_s$ , that has g-Delivered  $m'$  at line 26, and  $m'$  before  $m$ .

So there exists a process  $p_r$  that g-Delivers  $m$  at line 26, and not  $m'$  before  $m$ , and there exists a process  $p_s$  that g-Delivers  $m'$  at line 26, and not  $m$  before  $m'$ . To g-Deliver  $m$  at line 26, process  $p_r$  has received (line 20)  $n_{ack}$  messages of the type  $(k, g\_Deliver^k, pending^k, ACK)$ , such that  $m \in pending^k$ . Similarly, to g-Deliver  $m'$  at line 26, process  $p_s$  has received (line 20)  $n_{ack}$  messages of the type  $(k, g\_Deliver^k, pending^k, ACK)$ , such that  $m' \in pending^k$ .

By the condition of line 14, the conflicting messages  $m$  and  $m'$  cannot be in the same set  $pending^k$ , and therefore, there must exist  $2n_{ack}$  different processes that have sent  $(k, g\_Deliver^k, pending^k, ACK)$  at line 16 – a contradiction since  $n_{ack} > n/2$ .  $\square$

**Proposition 6.** (UNIFORM ORDER).

(Assumes  $n_{ack}, n_{chk} > n/2$ , and  $2n_{ack} + n_{chk} \geq 2n + 1$ .)  
If processes  $p_i$  and  $p_j$  both g-Deliver conflicting messages  $m$  and  $m'$ , then  $p_i$  and  $p_j$  g-Deliver  $m$  and  $m'$  in the same order.

*Proof.* (uses Lemmas 2, 3, 6, and 7) The proof is close the proof of the corresponding property of  $\mathcal{GB}$  (Proposition 2). The difference stems from the fact that Proposition 2 relies on Lemma 5, which does not hold for  $\mathcal{GB}+$  and has been replaced with Lemma 7.

Without loss of generality, assume that  $p_i$  g-Delivers  $m$  before  $m'$ . As in the proof of Proposition 2, the result holds immediately if  $p_i$  g-Delivers  $m$  in stage  $k$  and  $m'$  in stage  $k' > k$ . From Lemma 7 the result also holds if  $p_i$  g-Delivers  $m$  and  $m'$  before the Consensus. It remains to consider the following cases: (1)  $p_i$  g-Delivers  $m$  before Consensus and  $m'$  after Consensus, and (2)  $p_i$  g-Delivers  $m$  and  $m'$  after Consensus. In case (1), from Lemma 7,  $p_j$  cannot g-Deliver  $m'$  before Consensus. The rest of case (1) can be proved as in the proof of Proposition 2. Case (2) can be proved as in the proof of Proposition 2.  $\square$

**Proposition 7.** (VALIDITY).

(Assumes  $n_{ack}, n_{chk} > n/2$ ,  $max(n_{ack}, n_{chk})$  correct processes,  $2n_{ack} + n_{chk} \geq 2n + 1$ .)  
If a correct process  $p_i$  g-Broadcasts a message  $m$ , then  $p_i$  eventually g-Delivers  $m$ .

*Proof.* Similar to the proof of Proposition 3.  $\square$

**Proposition 8.** (UNIFORM INTEGRITY). For any message  $m$ , each process g-Delivers  $m$  at most once, and only if  $m$  was previously g-Broadcast by sender( $m$ ).

*Proof.* Similar to the proof of Proposition 4.  $\square$

**Theorem 2.** Assume that there are  $max(n_{ack}, n_{chk})$  correct processes,  $n_{ack}, n_{chk} > n/2$ , and  $2n_{ack} + n_{chk} \geq 2n + 1$ . The algorithm in Fig. 2 solves Generic Broadcast, or reduces Generic Broadcast to a sequence of Consensus problems.

*Proof.* Immediate from Propositions 5, 6, 7, and 8.  $\square$

## 6 Evaluation of the Generic Broadcast algorithms

### 6.1 Time complexity in good runs

To assess the cost of Generic Broadcast, we consider “good” runs (i.e., runs with no failures and no failure suspicions). We express the delivery cost of a message  $m$  in terms of the maximum network message delay  $\delta$  [1]. We show below that if Consensus is not needed,  $\mathcal{GB}$  and  $\mathcal{GB}+$  g-Deliver messages in  $2\delta$ . However, if Consensus is necessary, at least  $4\delta$  are needed. By comparison, known Atomic Broadcast algorithms, in the model considered in the paper, can A-Deliver messages in  $3\delta$ ,<sup>5</sup> which shows the potential benefit of Generic Broadcast over Atomic Broadcast: if the message conflict rate is low, our Generic Broadcast algorithms are an interesting alternative to Atomic Broadcast algorithms. However, if the message conflict rate is high our Generic Broadcast algorithms become less efficient than known Atomic Broadcast algorithms.

### 6.2 Time complexity of $\mathcal{GB}$ and $\mathcal{GB}+$

We evaluate now the time between the execution of g-Broadcast( $m$ ) and g-Deliver( $m$ ), in terms of  $\delta$ , the maximum message delay.

#### 6.2.1 Time complexity in the first phase

For the first phase, we can do the same analysis for  $\mathcal{GB}$  and  $\mathcal{GB}+$ . Consider  $\mathcal{GB}$ , and let a process  $p$  g-Broadcast some message  $m$  (line 7). Message  $m$  is first R-broadcast (line 8), and upon R-delivery of  $m$  at line 10 (in the absence of failures, this takes  $\delta$ ), every process sends an ACK message to all, with  $m \in pending^k$  (line 15). Upon reception of ACK messages from  $n_{ack}$  processes (max delay =  $\delta$ ),  $m$  is g-Delivered (line 31). So, the time complexity of  $\mathcal{GB}$  and  $\mathcal{GB}+$  for message delivery in the first phase is  $2\delta$ .

#### 6.2.2 Time complexity in the second phase

We discuss now the cost of  $\mathcal{GB}$  and  $\mathcal{GB}+$  when Consensus is needed. Time complexity is more difficult to evaluate here, as the result depends on the interleaving of concurrent events. We give for each algorithm the best-case and the worst-case figures. For Consensus, we assume the algorithm in [11] that has a time complexity of  $2\delta$  in good runs.

<sup>5</sup> An exception is the Optimistic Atomic Broadcast algorithm [9], which can deliver messages in  $2\delta$  if the *spontaneous total order property* holds – that is, if messages are “spontaneously” received in the same order.

*Best case for  $\mathcal{GB}$ .* At time  $t$ , let process  $p$  g-Broadcast message  $m$  (line 7). Message  $m$  is R-delivered at line 10 (at time  $t + \delta$ ). In the best case, upon R-delivery of  $m$ , every process detects a conflict with some other message  $m'$  (line 13), and sends a message of type CHK to all, with  $m \in \text{pending}^k$  (line 17). All processes receive the message at time  $t + 2\delta$  (line 18) and start Consensus. As Consensus costs  $2\delta$ , message  $m$  is g-Delivered at time  $t + 4\delta$ .

*Worst case for  $\mathcal{GB}$ .* Let  $p$  again g-Broadcast  $m$  at time  $t$ . At time  $t + \delta$  all processes have R-delivered  $m$ , but not all processes detect a conflict at that time. So, not all processes send immediately a message of type CHK. However, at least one process  $q$  detects a conflict with some message  $m'$  at time  $t + \delta$  (otherwise Consensus is not needed). If  $q$  has R-delivered  $m'$  at time  $t + \delta$ , then all processes R-deliver  $m'$  at time  $t + 2\delta$ , detect the conflict with  $m$ , and send the message of type CHK. So, all processes start Consensus at time  $t + 3\delta$ , and end Consensus at  $t + 5\delta$ . However, this analysis assumes that between  $t + \delta$  and  $t + 3\delta$ , Task 3 is not involved in another Consensus not related to message  $m$ , in which case, such an execution of Consensus would have to terminate first, adding  $2\delta$ . Thus, in the worst case the g-Delivery of  $m$  takes  $7\delta$ .

*Best case for  $\mathcal{GB}+$ .* At time  $t$ , let process  $p$  g-Broadcast message  $m$  (line 8). Message  $m$  is R-delivered at line 11 (at time  $t + \delta$ ). In the best case, upon R-delivery of  $m$ , every process detects a conflict with some other message  $m'$  (line 14), and sends a message of type CHK to all, with  $m \in \text{pending}^k$  (line 19). All processes receive the message at time  $t + 2\delta$  (line 27), and start Consensus. As Consensus costs  $2\delta$ , message  $m$  is g-Delivered at time  $t + 4\delta$ .

*Worst case for  $\mathcal{GB}+$ .* Let  $p$  again g-Broadcast  $m$  at time  $t$ . At time  $t + \delta$  all processes have R-delivered  $m$ , but not all processes detect a conflict at that time. So, not all processes send immediately a message of type CHK. However, at least one process  $q$  detects a conflict with some message  $m'$  at time  $t + \delta$  (otherwise Consensus is not needed). Upon detecting the conflict, process  $q$  sends a message of type CHK to all (line 19). Upon reception of this message (time  $t + 2\delta$ ), the processes that have not yet sent the message of type CHK do so, with  $m \in \text{pending}^k$  (line 30). These messages are received at time  $t + 3\delta$ . So, all processes start Consensus at time  $t + 3\delta$ , and end Consensus at  $t + 5\delta$ . As for the worst case of  $\mathcal{GB}$ , the analysis ignores that at any time, between  $t + \delta$  and  $t + 3\delta$ , Task 3 might be involved in another Consensus not related to message  $m$ , which adds  $2\delta$ . So, in the worst case the g-Delivery of  $m$  takes  $7\delta$ .

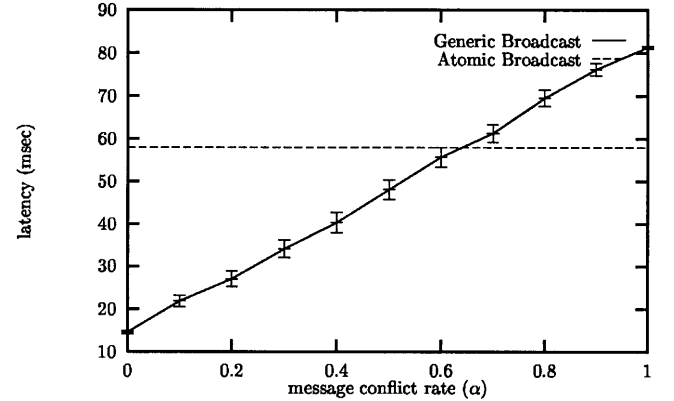
### 6.3 Generic Broadcast vs. Atomic Broadcast

Table 1 summarizes the time complexity of  $\mathcal{GB}$  and  $\mathcal{GB}+$ :  $2\delta$  in the first phase, and between  $4\delta$  (best case) and  $7\delta$  (worst case) if the second phase is needed. By comparison the time complexity of Atomic Broadcast is between  $3\delta$  (best case) and  $5\delta$  (worst case).<sup>6</sup>

<sup>6</sup> We consider Atomic Broadcast solved by reduction to Consensus [4] and the Consensus algorithm of [11].

**Table 1.** Generic Broadcast vs. Atomic Broadcast

Protocols	Best Case	Worst Case
$\mathcal{GB}$ and $\mathcal{GB}+$ : Phase I only	$2\delta$	–
Phase I and II	$4\delta$	$7\delta$
Atomic Broadcast	$3\delta$	$5\delta$



**Fig. 3.** Comparing Generic Broadcast to Atomic Broadcast ( $\alpha = 0$ : only non-conflicting messages are g-Broadcast;  $\alpha = 1$ : only conflicting messages are g-Broadcast)

### 6.4 Experimental validation

The results of Sect. 6.3 are confirmed by an experiment conducted with 10 processes ( $n = 10$ ) running on Sun’s Ultra-Sparc workstations interconnected by an Ethernet network (10 MBit/s) and communicating using TCP/IP (see Fig. 3). The experiment measures the cost of the “Best Case” of Table 1. Processes implement the  $\mathcal{GB}$  algorithm with  $n_{ack} = n_{chk} = 7$ . The vertical axis of Fig. 3 represents the time elapsed between the events g-Broadcast( $m$ ) and g-Deliver( $m$ ) at the sender of  $m$ . The horizontal axis represents the message conflict rate, that is, the ratio of the number of g-Broadcast messages that conflict to the total number of g-Broadcast messages.

Thus,  $\alpha = 0$  means that only non-conflicting messages were g-Broadcast, while  $\alpha = 1$  means that only conflicting messages were g-Broadcast. In other words,  $\alpha = 0$  measures the cost of the first phase of  $\mathcal{GB}$ , while  $\alpha = 1$  measures the cost of the first and the second phases. The Atomic Broadcast algorithm is the one mentioned in Sect. 6.3 (notice that this algorithm requires a majority of correct processes, i.e., 6). Experiments were repeated to build a confidence interval of 95%, and in each experiment, processes g-Broadcast messages at a constant rate. From Fig. 3, if less than 60% of the messages g-Broadcast conflict, the  $\mathcal{GB}$  algorithm can g-Deliver messages more efficiently than the Atomic Broadcast algorithm considered.

## 7 Conclusion

The paper has introduced the Generic Broadcast problem, whose definition is based on a conflict relation on the set of messages that are broadcast. The conflict relation can be derived from the semantics of the messages, and only conflicting messages have to be delivered by all processes in the

same order. As such, Generic Broadcast is a powerful message ordering abstraction, which includes Reliable and Atomic Broadcast as special cases. Generic Broadcast algorithms  $\mathcal{GB}$  and  $\mathcal{GB}+$  have been shown to be more efficient than Atomic Broadcast algorithms if message conflicts are not too frequent.

This paper, together with [1], show a time complexity vs. resilience tradeoff for Generic Broadcast algorithms. Our Generic Broadcast algorithms require  $f < n/3$  with a best case time complexity of  $2\delta$  (if messages do not conflict). In [1], the authors propose Generic Broadcast algorithms that require only  $f < n/2$ , with a time complexity of  $3\delta$  in the best case. So additional resilience increases the best time complexity. An interesting open question is whether there exist Generic Broadcast algorithms that can – in the best case – deliver messages in  $2\delta$ , and still require only a majority of correct processes.

*Acknowledgements.* We would like to thank the anonymous reviewers for their comments and suggestions that helped improve the paper.

## References

1. M.K. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg. Thrifty generic broadcast. In *Proceedings of the 14th International Symposium on Distributed Computing (DISC'2000)*, October 2000.
2. P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.
3. K.P. Birman and T.A. Joseph. Exploiting virtual synchrony in distributed systems. In *Proceedings of the 11th ACM Symposium on OS Principles*, pages 123–138, November 1987.
4. T.D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.
5. D. Cheriton and D. Skeen. Understanding the limitations of causally and totally ordered communication. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, Asheville (USA), December 1993.
6. M.J. Fischer, N.A. Lynch, and M.S. Paterson. Impossibility of distributed consensus with one faulty processor. *Journal of the ACM*, 32(2):374–382, 1985.
7. V. Hadzilacos and S. Toueg. Fault-tolerant broadcasts and related problems. In: *Distributed Systems*, chapter 5. Addison-Wesley, 2nd edition, 1993.
8. P.A. Jensen, N.R. Soparkar, and A.G. Mathur. Characterizing multicast orderings using concurrency control theory. In: *Proceedings of the 17th International Conference on Distributed Computing Systems (ICDCS'97)*, Baltimore (USA), May 1997.
9. F. Pedone. *The Database State Machine and Group Communication Issues*. PhD thesis, École Polytechnique Fédérale de Lausanne, Switzerland, December 1999. Number 2090.
10. F. Pedone and A. Schiper. Generic broadcast. In: *Proceedings of the 13th International Symposium on Distributed Computing (DISC'99, formerly WDAG)*, September 1999.
11. A. Schiper. Early consensus in an asynchronous system with a weak failure detector. *Distributed Computing*, 10(3):149–157, 1997.
12. F.B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.