Generic Broadcast

Fernando Pedone André Schiper

Département d'Informatique
Ecole Polytechnique Fédérale de Lausanne
1015 Lausanne, Switzerland
{Fernando.Pedone, Andre.Schiper@epfl.ch}

Abstract

Message ordering is a fundamental abstraction in distributed systems. However, usual ordering guarantees are purely "syntactic", that is, message "semantics" is not taken into consideration, despite the fact that in several cases, semantic information about messages leads to more efficient message ordering protocols. In this paper we define the *Generic Broadcast* problem, which orders the delivery of messages only if needed, based on the semantics of the messages. Semantic information about the messages is introduced in the system by a conflict relation defined over messages. We show that Reliable and Atomic Broadcast are special cases of Generic Broadcast, and propose an algorithm that solves Generic Broadcast efficiently. In order to assess efficiency, we introduce the concept of *deliver latency.*

1 Introduction

Message ordering is a fundamental abstraction in distributed systems. Total order, causal order, view synchrony, etc., are examples of widely used ordering guarantees. However, these ordering guarantees are purely "syntactic" in the sense that they do not take into account the "semantics" of the messages. Active replication for example (also called state machine approach [11]), relies on total order delivery of messages on the active replicated servers. By considering the semantics of the messages sent to active replicated servers, total order delivery may not always be needed. This is the case for example if we distinguish *read* messages from *write* messages sent to active replicated servers, since read messages do not need to be ordered with respect to other read messages. As message ordering has a cost, it makes sense to avoid ordering messages when not required.

In this paper we define the *Generic Broadcast* problem (defined by the primitives *g-Broadcast* and *g-Deliver*), which establishes a partial order on message delivery. Semantic information about messages is introduced in the system by a *conflict* relation defined over the set of messages. Roughly speaking, two messages m and m' have to be g-Delivered in the same order only if m and m' are conflicting messages. 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 the Generic Broadcast problem in which the conflict relation is empty. The Atomic Broadcast problem is another instance of the Generic Broadcast problem, in which all pair of messages conflict.

Any algorithm that solves Atomic Broadcast trivially solves any instance of Generic Broadcast (i.e., specified by a given conflict relation), even if ordering more messages than necessary. Thus, we define a Generic Broadcast algorithm to be *strict* if it only orders messages when necessary. The notion of strictness captures the intuitive idea that total order delivery of messages has a cost, and this cost should only be paid when necessary.

Although the notion of strictness adequately represents the idea behind a satisfactory solution to the Generic Broadcast problem, we show that strictness can be ensured by an algorithm as expensive as an Atomic Broadcast algorithm. Therefore we introduce the concept of *deliver latency* of a message to assess the cost of Generic Broadcast algorithms. Roughly speaking, the deliver latency of a message m is the number of communication steps between *g-Broadcast(m)* and *g-Deliver(m)*. We then give a strict Generic Broadcast algorithm that – in runs where messages do not conflict – ensures that the deliver latency of every message is always equal to 2 (Atomic Broadcast algorithms have at least deliver latency equal to 3).

The rest of the paper is structured as follows. Section 2 defines the Generic Broadcast problem. Section 3 defines the system model and introduces the concept of deliver latency. Section 4 presents a solution to the Generic Broadcast problem. Section 5 discusses related work, and Section 6 concludes the paper.

2 Generic Broadcast

2.1 Problem Definition

Generic Broadcast is defined by the primitives g-Broadcast and g-Deliver.[1] When a process p invokes g-Broadcast with a message m, we say that p g-Broadcasts m, and when p returns from the execution of g-Deliver with message m, we say that p g-Delivers m. Message m is taken from a set $\mathcal{M}$ to which all messages belong. Central to Generic Broadcast is the definition of a (symmetric) conflict relation on $\mathcal{M} \times \mathcal{M}$ denoted by $\mathcal{C}$ (i.e., $\mathcal{C} \subseteq \mathcal{M} \times \mathcal{M}$). If $(m, m') \in \mathcal{C}$ then we say that m and m' conflict. Generic Broadcast is specified by (1) a conflict relation $\mathcal{C}$ and (2) the following conditions:

[1]g-Broadcast has no relation with the GBCAST primitive defined in the Isis system [1].

gB-1 (VALIDITY). If a correct process g-Broadcasts a message m, then it eventually g-Delivers m.

gB-2 (AGREEMENT). If a correct process g-Delivers a message m, then all correct processes eventually g-Deliver m.

gB-3 (INTEGRITY). For any message m, every correct process g-Delivers m at most once, and only if m was previously g-Broadcast by some process.

gB-4 (PARTIAL ORDER). If correct processes p and q both g-Deliver messages m and m', and m and m' conflict, p g-Delivers m before m' if and only if q g-Delivers m before m'.

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

2.2 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 $\mathcal{C}_\emptyset$, where $\mathcal{C}_\emptyset = \emptyset$, and (2) the $\mathcal{M} \times \mathcal{M}$ conflict relation, denoted by $\mathcal{C}_{\mathcal{M}\times\mathcal{M}}$, where $\mathcal{C}_{\mathcal{M}\times\mathcal{M}} = \mathcal{M} \times \mathcal{M}$. In case (1) no pair of messages conflict, that is, the partial order property gB-4 imposes no constraint. This is equivalent to having only the conditions gB-1, gB-2 and gB-3, which is called *Reliable Broadcast* [4]. In case (2) any pair (m, m') of messages conflict, that is, the partial order property gB-4 imposes that all pairs of messages be ordered, which is called *Atomic Broadcast* [4]. 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.[2] Let $\mathcal{M}_{deposit}$ denote the set of messages that carry a *deposit* operation, and $\mathcal{M}_{withdraw}$ the set of messages that carry a *withdraw* operation. This leads to the following conflict relation $\mathcal{C}_{Account}$:

$$\mathcal{C}_{Account} = \{\ (m, m')\ :\ m \in \mathcal{M}_{withdraw} \textbf{ or } m' \in \mathcal{M}_{withdraw}\}.$$

[2]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.

Generic Broadcast with the $\mathcal{C}_{Account}$ conflict relation for broadcasting the invocation of deposit and withdraw operations to the replicated *Account* object defines a weaker ordering primitive than Atomic Broadcast (e.g., messages in $\mathcal{M}_{deposit}$ are not required to be ordered with each other), and a stronger ordering primitive than Reliable Broadcast (which imposes no order at all).

2.3 Strict Generic Broadcast Algorithm

From the specification it is obvious that any algorithm solving Atomic Broadcast also solves any instance of the Generic Broadcast problem defined by $\mathcal{C} \subseteq \mathcal{M} \times \mathcal{M}$. However, such a solution also orders messages that do not conflict. We are interested in a *strict* algorithm, that is, an algorithm that does not order two messages if not required, according to the conflict relation $\mathcal{C}$. The idea is that ordering messages has a cost (in terms of number of messages, number of communication steps, etc.) and this cost should be kept as low as possible. More formally, we define an algorithm that solves Generic Broadcast for a conflict relation $\mathcal{C} \subset \mathcal{M} \times \mathcal{M}$, denoted by $A_{\mathcal{C}}$, *strict* if it satisfies the condition below.

> (Strictness). Consider an algorithm $A_{\mathcal{C}}$, and let $\mathcal{R}^{NC}_{\mathcal{C}}$ be the set of runs of $A_{\mathcal{C}}$ in which no conflicting messages are g-Broadcast. Then there exists a run R in $\mathcal{R}^{NC}_{\mathcal{C}}$, such that in R at least two processes p_i and p_j g-Deliver two messages m and m' in a different order.

Informally, the strictness condition requires that algorithm $A_{\mathcal{C}}$ allow runs in which the g-Deliver of non conflicting messages is not totally ordered. However, even if $A_{\mathcal{C}}$ does not order messages, it can happen that total order is spontaneously ensured. So we cannot require violation of total order to be observed in every run: we require it in at least one run of $A_{\mathcal{C}}$.

2.4 A Trivial Strict Generic Broadcast Algorithm

In the following, we present a trivial strict Generic Broadcast algorithm, based on Atomic Broadcast. Every process p_i has a buffer BUF_i that can hold one message. Atomic Broadcast is defined by the primitives A-Broadcast and A-Deliver.

- *g-Broadcast(m)* is executed by calling *A-Broadcast(m)*;
- on executing *A-Deliver(m)* each process p_i does the following:[3]

```
    if $BUF_i$ is empty then
        store $m$ in $BUF_i$
```

[3]To simplify the presentation, this algorithm assumes that an even number of messages is g-Broadcast.

```
else
    let $m'$ be the message removed from $BUF_i$
    if $m$ and $m'$ do not conflict and $i$ is an odd number then
        g-Deliver(m); g-Deliver(m')
    else
        g-Deliver(m'); g-Deliver(m)
```

The drawback of the algorithm above is that it has the "cost" of an Atomic Broadcast algorithm. In the next sections, we present a Generic Broadcast algorithm that is "cheaper" than the algorithm above. In particular, we consider the deliver latency as the parameter to measure cost (defined in Section 3.2).

3 System Model and Definitions

3.1 Processes, Failures and Failure Detectors

We consider an asynchronous system composed of n processes $\Pi = \{p_1, \ldots, p_n\}$. Processes communicate by message passing. A process can only fail by crashing (i.e., we do not consider Byzantine failures). Processes are connected through reliable channels, defined by the two primitives *send*(m) and *receive*(m). We assume that the asynchronous system is augmented with failure detectors allowing to solve Consensus (e.g., the class of failure detector $\Diamond S$ allows Consensus to be solved if $f < n/2$) [2].

3.2 Deliver Latency

In the following, we introduce the deliver latency as a measure of the efficiency of algorithms solving a Broadcast problem (defined by the primitives α-Broadcast and α-Deliver). The deliver latency is a variation of the Latency Degree introduced [10], which is based on modified Lamport's clocks [7].

- a *send* event and a *local* event on a process p_i do not modify p_i's local clock,
- let $ts(send(m))$ be the timestamp of the $send(m)$ event, and $ts(m)$ the timestamp carried by message m: $ts(m) \overset{\text{def}}{=} ts(send(m)) + 1$,
- the timestamp of $receive(m)$ on a process p_i is the maximum between $ts(m)$ and p_i's current clock value.

The deliver latency of a message m α-Broadcast in a run R of an algorithm A solving a Broadcast problem, denoted by $dl^R(m)$, is defined as the difference between the largest timestamp of all α-Deliver(m) events (at most one per process) in run R and the timestamp of the α-Broadcast(m) event in run R.

Let set^R_m be the set of processes that α-Deliver message m in run R. The deliver latency of m in run R is formally defined as

$$dl^R(m) \overset{\text{def}}{=} \underset{p \in set^R_m}{MAX} \; (ts(\alpha\text{-Deliver}_p(m)) - ts(\alpha\text{-Broadcast}(m))).$$

For example, consider a broadcast algorithm where a process p, willing to broadcast a message m, sends m to all processes, each process q on receiving m sends an acknowledge message $ACK(m)$ to all processes, and as soon as q receives n_{ack} $ACK(m)$ messages, q delivers m. Let R be a run of this algorithm where only m is broadcast. It follows that $dl^R(m) = 2$.

4 Solving Generic Broadcast

4.1 Overview of the Algorithm

Provided that the number of correct processes is at least $max(n_{ack}, n_{chk})$ (the definitions of n_{ack} and n_{chk} are presented next), Algorithm 1 (see page 9) solves Generic Broadcast for any conflict relation $\mathcal{C}$. Processes executing Algorithm 1 progress in a sequence of local stages numbered $1, 2, ..., k$. Each stage is terminated by a Consensus to decide on two sets of messages, denoted by $NCmsgSet^k$ (NC stands for Non Conflicting) and $CmsgSet^k$ (C stands for Conflicting). The set $NCmsgSet^k \cup CmsgSet^k$ is the set of messages that are g-Delivered in stage k. All messages in $NCmsgSet^k$ are g-Delivered by all processes before all messages in $CmsgSet^k$. The set $NCmsgSet^k$ does not contain conflicting messages, while messages in $CmsgSet^k$ may conflict. Messages in $CmsgSet^k$ are g-Delivered in some deterministic order. Process p starts stage $k + 1$ once it has g-Delivered all messages in $CmsgSet^k$. Messages in $NCmsgSet^k$ may be g-Delivered by process p in stage k before p executes the k-th Consensus. Such messages are g-Delivered without the *cost* of a Consensus execution. Furthermore, Algorithm 1 satisfies the following two properties:

(a) If m and m' are two conflicting messages then they are g-Delivered either (1) in different stages, or (2) in the same stage k, but at most one of them is in $NCmsgSet^k$.

(b) If message m is g-Delivered by some process p in stage k, then no process g-Delivers message m in stage k', $k \neq k'$.

Property (a) is ensured by having processes exchange ACK messages among each other before g-Delivering a message at line 36, and property (b) is ensured by having processes exchange

CHK (i.e., checking) messages whose role is to compute the initial value of each process p_i before starting Consensus execution at line 21. More specifically, a process needs n_{ack} ACK messages to g-Deliver a message at line 36, and n_{chk} CHK messages to define its initial value before starting the k-th Consensus execution at line 21. Properties (a) and (b) are guaranteed if n_{ack} and n_{chk} are such that

$$n_{ack} \geq (n+1)/2, \text{ and} \tag{1}$$

$$2n_{ack} + n_{chk} \geq 2n + 1. \tag{2}$$

The proof is given in Section 4.3. Intuitively the idea is as follows (see Figure 1). Let $ackPSet^k(m)$ be a set containing processes that have sent an ACK message for message m in stage k. Condition (1) ensures that for any two messages m and m', if $|ackPSet^k(m)| \geq n_{ack}$ and $|ackPSet^k(m')| \geq n_{ack}$, then $ackPSet^k(m) \cap ackPSet^k(m')$ is non-empty. Let $chkPSet^k$ be the set containing processes that send a CHK message in stage k. Condition (2) ensures that for any two messages m and m', if $|ackPSet^k(m)| \geq n_{ack}$ and $|chkPSet^k| \geq n_{chk}$, then $ackPSet^k(m) \cap chkPSet^k$, denoted by $PSet^k(m)$, contains a majority of processes from $chkPSet^k$.

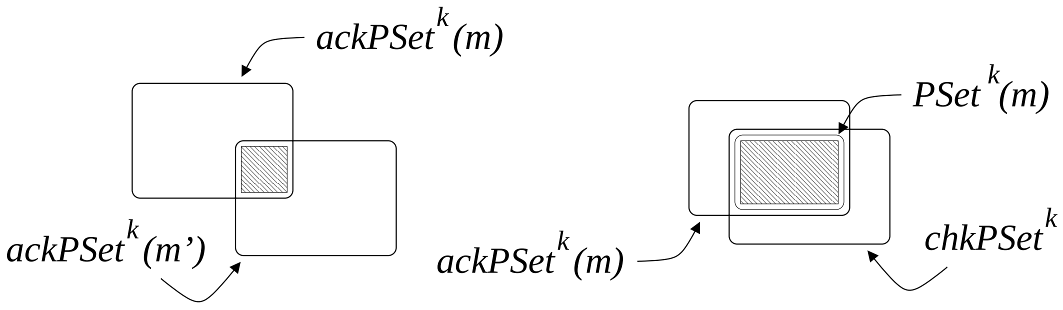

Figure 1: Acknowledge and checking sets

From conditions (1) and (2), and the fact that Algorithm 1 requires $max(n_{ack}, n_{chk})$ correct processes, we can determine the minimal number of correct processes for solving Generic Broadcast with Algorithm 1 (which happens when $n_{ack} = n_{chk}$) to be $(2n + 1)/3$ processes.

4.2 The Generic Broadcast Algorithm

All tasks in Algorithm 1 execute concurrently, and Task 3 has two entry points (lines 12 and 31). Process p in stage k manages the following sets.

- $R\_delivered_p$: contains all messages R-delivered by p up to the current time,

- $G\_delivered_p$: contains all messages g-Delivered by p in all stages $k' < k$,
- $pending_p^k$: contains every message m such that p has sent an ACK message for m in stage k up to current time, and
- $localNCg\_Deliver_p^k$: is the set of non conflicting messages that are g-Delivered by p in stage k, up to the current time (and before p executes the k-th Consensus).

When p wants to g-Broadcast message m, p executes *R-broadcast*(m) (line 8). After R-delivering a message m, the actions taken by p depend on whether m conflicts or not with some other message m' in $R\_delivered_p \setminus G\_delivered_p$.

No conflict. If no conflict exists, then p includes m in $pending_p^k$ (line 14), and sends an ACK message to all processes, acknowledging the R-deliver of m (line 15). Once p receives n_{ack} ACK messages for a message m (line 31), p includes m in $localNCg\_Deliver_p^k$ (line 35) and g-Delivers m (line 36).

Conflict. In case of conflict, p starts the terminating procedure for stage k. Process p first sends a message of the type $(k, pending_p^k, CHK)$ to all processes (line 17), and waits the same information from n_{chk} processes (line 18). Then p builds the set $majMSet_p^k$ (line 20)[4]. It can be proved that $majMSet_p^k$ contains every message m such that for any process q, $m \in localNCg\_Deliver_q^k$. Then p starts consensus (line 21) to decide on a pair $(NCmsgSet^k, CmsgSet^k)$ (line 22). Once the decision is made, process p first g-Delivers (in any order) the messages in $NCmsgSet^k$ that is has not g-Delivered yet (lines 23 and 25), and then p g-Delivers (in some deterministic order) the messages in $CmsgSet^k$ that it has not g-Delivered yet (lines 24 and 26). After g-Delivering all messages decided in Consensus execution k, p starts stage $k+1$ (lines 28-30).

4.3 Proof of Correctness

The correctness of the Generic Broadcast algorithm presented in Section 4.2 follows from Propositions 1 (Agreement), 2 (Partial Order), 3 (Validity), and 4 (Integrity) that are given in the Appendix. Propositions 1 and 2 follow from the Lemmata 2 and 3 below. Lemma 2 follows from Lemma 1. All proofs are in the Appendix.

Lemma 1 relates the sets $ackPSet^k$, $chkPSet^k$, and $PSet^k$ (see Figure 1). It states that, provided that $2n_{ack} + n_{chk} \geq 2n + 1$ (Condition 2, page 7), any intersection between $ackPSet^k$ and $chkPSet^k$ contains a set $PSet^k$ which contains a majority of the elements in $chkPSet^k$.

[4] $majMSet_p^k = \{m \mid |PSet_p^k(m)| \geq (n_{chk}+1)/2\}$

Algorithm 1 Generic Broadcast

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

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

8:    R-broadcast($m$)

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

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

12:   when $(R\_delivered \setminus G\_delivered) \setminus pending^k \neq \emptyset$     {Task 3}
13:      if [ for all $m, m' \in R\_delivered \setminus G\_delivered,\ m \neq m' : (m, m') \notin Conflict$ ] 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 $q : p$ received ($\underline{k}, pending^k_q, \underline{CHK}$) from $q$ ]
19:         #Define $chkPSet^k(m) = \{q : p$ received $(\underline{k}, pending^k_q, \underline{CHK})$ from $q$ and $m \in pending^k_q\}$
20:         $majMSet^k \leftarrow \{m : |\,chkPSet^k(m)\,| \geq \lceil (n_{chk}+1)/2 \rceil\}$
21:         $propose(k, (majMSet^k, (R\_delivered \setminus G\_delivered) \setminus majMSet^k))$
22:         wait until $decide(\underline{k}, (NCmsgSet^k, CmsgSet^k))$
23:         $NCg\_Deliver^k \leftarrow (NCmsgSet^k \setminus localNCg\_Deliver^k) \setminus G\_delivered$
24:         $Cg\_Deliver^k \leftarrow CmsgSet^k \setminus G\_delivered$
25:         g-Deliver messages in $NCg\_Deliver^k$ in any order
26:         g-Deliver messages in $Cg\_Deliver^k$ using some deterministic order
27:         $G\_delivered \leftarrow (localNCg\_Deliver^k \cup NCg\_Deliver^k \cup Cg\_Deliver^k) \cup G\_delivered$
28:         $k \leftarrow k + 1$
29:         $pending^k \leftarrow \emptyset$
30:         $localNCg\_Deliver^k \leftarrow \emptyset$

31:   when receive($\underline{k}, pending^k_q, \underline{ACK}$) from $q$
32:      #Define $ackPSet^k(m) = \{q : p$ received $(\underline{k}, pending^k_q, \underline{ACK})$ from $q$ and $m \in pending^k_q\}$
33:      $ackMSet^k \leftarrow \{m : |ackPSet^k(m)| \geq n_{ack}\}$
34:      $localNCmsgSet^k \leftarrow ackMSet^k \setminus (G\_delivered \cup NCmsgSet^k)$
35:      $localNCg\_Deliver^k \leftarrow localNCg\_Deliver^k \cup localNCmsgSet^k$
36:      g-Deliver all messages in $localNCmsgSet^k$ in any order
```

Lemma 1 *Let $ackPSet^k(m)$ be a set containing n_{ack} processes that execute send$(k, pending^k, ACK)$ (line 15) in stage k such that $m \in pending^k$, and let $chkPSet^k$ be the set of processes from which some process p receives n_{chk} messages of the type $(k, pending^k, CHK)$ in stage k (line 18). If $2n_{ack} + n_{chk} \geq 2n + 1$, then there are at least $(n_{chk} + 1)/2$ processes in $(chkPSet^k \cap ackPSet^k(m))$.*

Lemma 2 states that any message g-Delivered by some process q during stage k, before q executes Consensus in stage k will be included in the set $NCmsgSet^k$ decided by Consensus k.

Lemma 2 *For any two processes p and q, and all $k \geq 1$, if p executes decide$(k, (NCmsgSet^k, -))$, then $localNCg\_Deliver^k_q \subseteq NCmsgSet^k$.*

Lemma 3 states that the set *pending*^{k} does not contain conflicting messages.

Lemma 3 *For any process p, and all $k \geq 1$, if messages m and m' are in $pending^k_p$, then m and m' do not conflict.*

4.4 Strictness and Cost of the Generic Broadcast Algorithm

Proposition 5 states that the Generic Broadcast algorithm of Section 4.2 is a strict implementation of Generic Broadcast.

Proposition 5 *Algorithm 1 is a strict Generic Broadcast algorithm.*

We now discuss the cost of our Generic Broadcast algorithm. Our main result is that for messages that do not conflict, the Generic Broadcast algorithm can deliver messages with a deliver latency equal to 2, while for messages that conflict, the deliver latency is at least equal to 4. Since known Atomic Broadcast algorithms deliver messages with a deliver latency of at least 3,[5] this results shows the tradeoff of the Generic Broadcast algorithm: if messages conflict frequently, our Generic Broadcast algorithm may become less efficient than an Atomic Broadcast algorithm, while if conflicts are rare, then our Generic Broadcast algorithm leads to smaller costs compared to Atomic Broadcast algorithms.

Propositions 6 and 7 assess the cost of the Generic Broadcast algorithm when messages do not conflict. In order to simplify the analysis of the deliver latency, we concentrate our results on runs with one message (although the results can be extended to more general runs). Proposition 6 defines a lower bound on the deliver latency of the algorithm, and Proposition 7

[5]An exception is the Optimistic Atomic Broadcast algorithm [8], which can deliver messages with deliver latency equal to 2 if the *spontaneous total order property* holds.

shows that this bound can be reached in runs where there are no process failures nor failure suspicions. We consider a particular implementation of Reliable Broadcast that appears in [2].[6]

Proposition 6 *Assume that Algorithm 1 uses the Reliable Broadcast implementation presented in [2]. If $\mathcal{R}_\mathcal{C}$ is a set of runs generated by Algorithm 1 such that m is the only message g-Broadcast and g-Delivered in runs in $\mathcal{R}_\mathcal{C}$, then there is no run R in $\mathcal{R}_\mathcal{C}$ where $dl^R(m) < 2$.*

Proposition 7 *Assume that Algorithm 1 uses the Reliable Broadcast implementation presented in [2]. If $\mathcal{R}_\mathcal{C}$ is a set of runs generated by Algorithm 1, such that in runs in $\mathcal{R}_\mathcal{C}$, m is the only message g-Broadcast and g-Delivered, and there are no process failures nor failure suspicions, then there is a run R in $\mathcal{R}_\mathcal{C}$ where $dl^R(m) = 2$.*

The results that follow define the behaviour of the Generic Broadcast algorithm in runs where conflicting messages are g-Broadcast. Proposition 8 establishes a lower bound for cases where messages conflict, and Proposition 9 shows that the *best* case with conflicts can be reached when there are no process failures nor failure suspicions.

Proposition 8 *Assume that Algorithm 1 uses the Reliable Broadcast implementation presented in [2], and the Consensus implementation presented in [10]). Let $\mathcal{R}_\mathcal{C}$ be a set of runs generated by Algorithm 1, such that m and m' are the only messages g-Broadcast and g-Delivered in $\mathcal{R}_\mathcal{C}$. If m and m' conflict, then there is no run R in $\mathcal{R}_\mathcal{C}$ where $dl^R(m) < 4$ and $dl^R(m') < 4$.*

Proposition 9 *Assume that Algorithm 1 uses the Reliable Broadcast implementation presented in [2], and the Consensus implementation presented in [10]). Let $\mathcal{R}_\mathcal{C}$ be a set of runs generated by Algorithm 1, such that m and m' are the only messages g-Broadcast and g-Delivered in $\mathcal{R}_\mathcal{C}$, and there are no process failures nor failure suspicions. If m and m' conflict, then there is a run R in $\mathcal{R}_\mathcal{C}$ where m is g-Delivered before m' and $dl^R(m) = 2$ and $dl^R(m') = 4$.*

5 Related Work

Group communication aim at extending traditional one-to-one communication, which is insufficient in many settings. One-to-many communication is typically needed to handle replication (replicated data, replicated objects, etc.). Classical techniques to manage replicated data are based on voting and quorum systems (e.g., [3, 5, 6] to cite a few). Early quorum systems distinguish read operations from write operations in order to allow for concurrent read operations. These ideas have been extended to abstract data types in [5]. Increasing concurrency, without compromising the strong consistency guarantees on replicated data, is a standard way to

[6]Whenever a process p wants to R-broadcast a message m, p sends m to all processes. Once a process q receives m, if $q \neq p$ then q sends m to all processes, and, in any case, q R-delivers m.

increase the performances of the system. Lazy replication [9] is another approach that aims at increasing the performances by reducing the cost of replication. Lazy replication also distinguishes between read and write operations, and relaxes the requirement of total order delivery of read operations. Consistency is ensured at the cost of managing timestamps outside of the set of replicated servers; these timestamps are used to ensure Causal Order delivery on the replicated servers.

Our approach also aims at increasing the performances of replication by increasing concurrency in the context of group communications. Similarly to quorum systems, our Generic Broadcast algorithm allows for concurrency that is not possible with traditional replication techniques based on Atomic Broadcast. From this perspective, our work can be seen as a way to integrate group communications and quorum systems. There is even a stronger similarity between quorum systems and our Generic Broadcast algorithm. Our algorithm is based on two sets: an acknowledgement set and a checking set.[7] These sets play a role similar to quorum systems. However, quorum systems require weaker conditions to keep consistency than the condition required by the acknowledgement and checking sets.[8] Although the reason for this discrepancy is very probably related to the guarantees offered by quorum systems, the question requires further investigation.

6 Conclusions

The paper has introduced the Generic Broadcast problem, which is defined based on a conflict relation on the set of messages. The notion of conflict can be derived from the semantic of the messages. 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. The advantage of Generic Broadcast over Atomic Broadcast is a cost issue, where cost is defined by the notion of deliver latency of messages.

On a different issue, our Generic Broadcast algorithm uses mechanisms that have similarities with quorum systems. As future work it would be interesting to investigate this point to better understand the differences between replication protocols based on group communication (e.g., Atomic Broadcast, Generic Broadcast) and replication protocols based on quorum systems.

Finally, as noted in Section 4.1, our Generic Broadcast algorithm requires at least $(2n+1)/3$ correct processes. Such a condition is usual in the context of Byzantine failures, but rather surprising in the context of crash failures.

[7]Used respectively for g-Delivering non-conflicting messages during a stage, and determining non-conflicting messages g-Delivered at the termination of a stage.

[8]Let n_r be the size of a read quorum, and n_w the size of a write quorum. Quorum systems usually requires that $n_r + n_w \geq n + 1$.

References

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

[2] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.

[3] D.K. Gifford. Weighted Voting for Replicated Data. In *Proceedings of the 7th Symposium on Operating Systems Principles*, pages 150–159, December 1979.

[4] V. Hadzilacos and S. Toueg. Fault-Tolerant Broadcasts and Related Problems. In *Distributed Systems*, chapter 5. Addison Wesley, second edition, 1993.

[5] M. Herlihy. A Quorum-Consensus Replication Method for Abstract Data Types. *ACM Transactions on Computer Systems*, 4(1):32–53, February 1986.

[6] S. Jajodia and D. Mutchler. Dynamic Voting. In *Proc. of the ACM SIGMOD Int. Conference on Management of Data*, pages 227–238, May 1987.

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

[8] F. Pedone and A. Schiper. Optimistic Atomic Broadcast. In *Proc. of 12th International Symposium on Distributed Computing*, pages 318–332, September 1998.

[9] S. Ghemawat R. Ladin, B. Liskov. Providing High Availability Using Lazy Replication. *ACM Transactions on Computer Systems*, 10(4):360–391, November 1992.

[10] A. Schiper. Early consensus in an asynchronous system with a weak failure detector. *Distributed Computing*, 10(3):149–157, 1997.

[11] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

Appendix - Proofs

Lemma 1 *Let $ackPSet^k(m)$ be a set containing n_{ack} processes that execute send$(k, pending^k, ACK)$ (line 15) in stage k such that $m \in pending^k$, and let $chkPSet^k$ be the set of processes from which some process p receives n_{chk} messages of the type $(k, pending^k, CHK)$ in stage k (line 18). If $2n_{ack} + n_{chk} \geq 2n + 1$, then there are at least $(n_{chk} + 1)/2$ processes in $(chkPSet^k \cap ackPSet^k(m))$.*

Proof: It follows that $|ackPSet(m)^k \cap chkPSet^k| \geq (n_{chk} + 1)/2$. Assume for a contradiction that $2n_{ack} + n_{chk} \geq 2n + 1$, and $|ackPSet(m)^k \cap chkPSet^k| < (n_{chk} + 1)/2$. We define $intSet^k = ackPSet(m)^k \cap chkPSet^k$, $ackOnlySet^k = ackPSet(m)^k \setminus intSet^k$, and $chkOnlySet^k = chkPSet^k \setminus intSet^k$. Therefore, (1) $ackPSet(m)^k = ackOnlySet^k \cup intSet^k$, and (2) $chkPSet^k = chkOnlySet^k \cup intSet^k$. Let $n_{ackOnly} = |ackOnlySet^k|$, $n_{chkOnly} = |chkPSet^k|$, and $n_{int} = |intSet^k|$.

From (1) and $ackOnlySet^k \cap intSet^k = \emptyset$, we have that (3) $n_{ack} = n_{ackOnly} + n_{int}$, and from (2) and $chkOnlySet^k \cap intSet^k = \emptyset$, we have that (4) $n_{chk} = n_{chkOnly} + n_{int}$. Substituting (3) and (4) in the hypothesis $2n_{ack} + n_{chk} \geq 2n + 1$, we have that (5) $3n_{int} \geq 2n + 1 - (2n_{ackOnly} + n_{chkOnly})$. Since $n_{int} < (n_{chk} + 1)/2$, (6) $n_{int} < n_{chkOnly} + 1$. From (5) and (6), $n_{ackOnly} + n_{chkOnly} > n + 1$. However, $ackOnlySet^k \cap chkOnlySet^k = \emptyset$, and so $n_{ackOnly} + n_{chkOnly} \leq n$, a contradiction that concludes the proof. □

Lemma 2 *For any two processes p and q, and all $k \geq 1$, if p executes decide$(k, (NCmsgSet^k, -))$, then $localNCg\_Deliver_q^k \subseteq NCmsgSet^k$.*

Proof: Let m be a message in $localNCg\_Deliver_q^k$. We first show that if p executes the statement *propose*$(k, majMSet_p^k, -))$, then $m \in majMSet_p^k$. Since $m \in localNCg\_Deliver_q^k$, q must have received n_{ack} messages of the type $(k, pending^k, ACK)$ (line 31) such that $m \in pending^k$. Thus, there are n_{ack} processes that sent m to all processes in the *send*$(-)$ statement at line 15. From Lemma 1, $m \in majMSet_p^k$. Therefore, for every process q that executes *propose*$(k, (majMSet_q^k, -))$, $m \in majMSet_q^k$. Let $(NCmsgSet^k, -)$ be the value decided on Consensus execution k. By the uniform validity of Consensus, there is a process r that executed *propose*$(k, (majMSet_r^k, -))$ such that $NCmsgSet^k = majMSet_r^k$, and so, $m \in NCmsgSet^k$. □

Lemma 3 *For any process p, and all $k \geq 1$, if messages m and m' are in $pending_p^k$, then m and m' do not conflict.*

Proof: Assume for a contradiction that there is a process p, and some $k \geq 1$ such that m and m' are in $pending_p^k$, and m and m' conflict. Since m and m' are in $pending_p^k$, p must have R-delivered m and m'. Without loss of generality, assume that p first R-delivers m and then m'. Thus, there is a time after p R-delivers m' such that p evaluates the *if* statement at line 13, and $m' \in R\_delivered_p$, $m' \notin G\_delivered_p$, and $m' \notin pending_p^k$. When this happens, $m \in R\_delivered_p$ (by the hypothesis m is R-delivered before m'), and $m \notin G\_delivered_p$ (if $m \in G\_delivered$, from lines 27-29 m and m' cannot be both in $pending_p^k$). Therefore, m and m' are in $R\_delivered \setminus G\_delivered$, the test at line 13 evaluates false, and m' is not included in $pending_p^k$, a contradiction that concludes the proof. $\square$

Lemma 4 *For any two correct processes p and q, and all $k \geq 1$:*

(1) If p executes send$(k, -, CHK)$, then q eventually executes send$(k, -, CHK)$.

(2) If p executes propose$(k, -)$, then q eventually executes propose$(k, -)$.

(3) If p g-Delivers messages in $NCg\_Deliver_p^k \cup Cg\_Deliver_p^k$, then

(3.1) q also g-Delivers messages in $NCg\_Deliver_q^k \cup Cg\_Deliver_q^k$, and

(3.2) $localNCg\_Deliver_p^k \cup NCg\_Deliver_p^k = localNCg\_Deliver_q^k \cup NCg\_Deliver_q^k$ and $Cg\_Deliver_p^k = Cg\_Deliver_q^k$.

PROOF: The proof is by simultaneous induction on (1), (2) and (3). (BASIC STEP.) For $k = 1$, we first show that if p executes *send*$(1, -, CHK)$ (line 17), then q also executes *send*$(1, -, CHK)$. If p executes *send*$(1, -, CHK)$, then p has R-delivered two messages, m and m', that conflict. From the agreement of R-broadcast, q also R-delivers m and m'. Assume that q first R-delivers m, and then m'. Since initially $G\_delivered_q = \emptyset$, there is a time when m and m' are in $R\_delivered_q \setminus G\_delivered_q$, and from Lemma 3, either m is not in $pending_q^1$, or neither m nor m' are in $pending_q^1$. Thus q eventually executes *send*$(1, -, CHK)$ (line 17). Since there are n_{chk} processes correct that also execute *send*$(1, -, CHK)$, q eventually receives n_{chk} messages of the type $(1, -, CHK)$ (line 18), and so, if p executes *propose*$(1, -)$ then q also executes *propose*$(1, -)$.

We now consider that p g-Delivers messages in $NCg\_Deliver_p^1 \cup Cg\_Deliver_p^1$. Before executing *decide*$(1, (NCmsgSet_p^1, CmsgSet_p^1))$, p executes *propose*$(1, -)$. By the first part of the lemma, q also executes *propose*$(1, -)$. By termination and uniform integrity of Consensus, q eventually executes *decide*$(1, -)$ and does it exactly once. We show that (a) $localNCg\_Deliver_p^1 \cup NCg\_Deliver_p^1 = localNCg\_Deliver_q^1 \cup NCg\_Deliver_q^1$, and (b) $Cg\_Deliver_p^1 = Cg\_Deliver_q^1$.

(a) From the algorithm (line 23), $NCg\_Deliver_p^1 = (NCmsgSet_p^1 \setminus localNCg\_Deliver_p^1) \setminus G\_delivered_p$. Initially both $G\_delivered_p$ and $G\_delivered_q$ are empty, and it follows that $localNCg\_Deliver_p^1 \cup NCg\_Deliver_p^1 = localNCg\_Deliver_p^1 \cup (NCmsgSet_p^1 \setminus localNCg\_Deliver_p^1)$. From Lemma 2, $localNCg\_Deliver_p^1 \subseteq NCmsgSet_p^1$, and so, $localNCg\_Deliver_p^1 \cup (NCmsgSet_p^1 \setminus localNCg\_Deliver_p^1) = NCmsgSet_p^1$, and also

$localNCg\_Deliver^1_q \cup (NCmsgSet^1_q \setminus localNCg\_Deliver^1_q) = NCmsgSet^1_q$. By agreement of Consensus, $NCmsgSet^1_p = NCmsgSet^1_q$, and it follows that $localNCg\_Deliver^1_p \cup NCg\_Deliver^1_p = localNCg\_Deliver^1_q \cup NCg\_Deliver^1_q$.

(b) From the algorithm (line 24), $Cg\_Deliver^1_p = CmsgSet^1_p \setminus G\_delivered_p$. Since initially $G\_delivered_p$ and $G\_delivered_q$ are empty, $Cg\_Deliver^1_p = CmsgSet^1_p$, and $Cg\_Deliver^1_q = CmsgSet^1_q$. By agreement of Consensus, for every p and q, $CmsgSet^1_p = CmsgSet^1_q$, and so, $Cg\_Deliver^1_p = Cg\_Deliver^1_q$.

(Inductive step.) Assume that the Lemma holds for all $k, 1 \leq k < l$. We proceed by first showing that if p executes $send(l, -, CHK)$ (line 17), then q also executes $send(l, -, CHK)$. If p executes $send(l, -, CHK)$, then from line 13, there exist two messages m and m' that conflict, and a time when m and m' are in $R\_delivered_p \setminus G\_delivered_p$, and $m \notin pending^l_p$. Since m and m' are not in $G\_delivered_p$, m and m' are not in $\cup_{i=1}^{k}(localNCg\_Deliver^i_p \cup NCg\_Deliver^i_p \cup Cg\_Deliver^i_p)$. By the induction hypothesis, m and m' are not in $\cup_{i=1}^{k}(localNCg\_Deliver^i_q \cup NCg\_Deliver^i_q \cup Cg\_Deliver^i_q)$. By the agreement property of R-broadcast, eventually m and m' belong to $R\_delivered_q$. From Lemma 3, and the fact that m and m' conflict, there is a time after which q g-Delivers all messages in $\cup_{i=1}^{k}(localNCg\_Deliver^i_q \cup NCg\_Deliver^i_q \cup Cg\_Deliver^i_q)$ such that there are two messages m and m' in $R\_delivered_q \setminus G\_delivered_q$, and m and m' are not both in $pending^l_q$. Thus, q eventually executes $send(l, -, CHK)$. Since there are at least n_{chk} processes correct that execute $send(l, -, CHK)$, q eventually receives n_{chk} messages of the type $(l, -, CHK)$ (line 18), and so, if p executed $propose(l, -)$, q also executes $propose(l, -)$.

We now consider that p g-Delivers messages in $NCg\_Deliver^l_p \cup Cg\_Deliver^l_p$. Before executing $decide(l, (NCmsgSet^l_p, CmsgSet^l_p))$, p executes $propose(l, -)$. By part (1) of the lemma, q also executes $propose(l, -)$. By the termination and agreement properties of Consensus, q eventually executes $decide(l, -)$ exactly once. We show next that (a) $localNCg\_Deliver^l_p \cup NCg\_Deliver^l_p = localNCg\_Deliver^l_p \cup NCg\_Deliver^l_q$, and (b) $Cg\_Deliver^l_p = Cg\_Deliver^l_q$.

(a) From the algorithm (line 23), $NCg\_Deliver^l_p = (NCmsgSet^l_p \setminus localNCg\_Deliver^l_p) \setminus G\_delivered_p$, and from Lemma 2, $localNCg\_Deliver^l_p \subseteq NCmsgSet^l_p$. It follows that $localNCg\_Deliver^l_p \cup NCg\_Deliver^l_p = NCmsgSet^l_p - G\_delivered_p$. By agreement of Consensus, $NCmsgSet^l_p = NCmsgSet^l_q$. From the algorithm, it can be shown that $G\_delivered = \cup_{i=1}^{k}(localNCg\_Deliver^i \cup NCg\_Deliver^i \cup Cg\_Deliver^i)$, and from the induction hypothesis, for all $1 \leq k < l : \cup_{i=1}^{k}(localNCg\_Deliver^i_p \cup NCg\_Deliver^i_p \cup Cg\_Deliver^i_p) = \cup_{i=1}^{k}(localNCg\_Deliver^i_q \cup NCg\_Deliver^i_q \cup Cg\_Deliver^i_q)$, and so, $G\_delivered_p = G\_delivered_q$. Therefore, $localNCg\_Deliver^l_p \cup NCg\_Deliver^l_p = localNCg\_Deliver^l_p \cup NCg\_Deliver^l_q$.

(b) From the algorithm (line 24), $Cg\_Deliver^l_p = CmsgSet^l_p \setminus G\_delivered_p$. When line 24 is evaluated, $G\_delivered_p = \cup_{i=1}^{k}(localNCg\_Deliver_p^i \cup NCg\_Deliver_p^i \cup Cg\_Deliver_p^i)$, and it follows from the induction hypothesis that $G\_delivered_p = G\_delivered_q$. By the agreement of Consensus, $CmsgSet_p^l = CmsgSet_q^l$, and thus, $Cg\_Deliver_p^l = Cg\_Deliver_q^l$. $\Box$

Proposition 1 (Agreement). *Let $f < max(n_{ack}, n_{chk})$. If a correct process p g-Delivers a message m, then every correct process q eventually g-Delivers m.*

Proof: Consider that p has g-Delivered message m in stage k. We show that q also g-Delivers m in stage k. There are two cases to consider: (a) p executes Consensus in stage k, and (b) p never executes Consensus in stage k.

(a) From Lemma 4, q also executes Consensus in stage k. Since p g-Delivers m in stage k, $m \in localNCg\_Deliver_p^k \cup NCg\_Deliver_p^k \cup Cg\_Deliver_p^k$, and so, $m \in localNCg\_Deliver_q^k \cup NCg\_Deliver_q^k \cup Cg\_Deliver_q^k$. Thus, q either g-Delivers m at line 36 (in which case $m \in localNCg\_Deliver_q^k$), or at line 25 (in which case $m \in NCg\_Deliver_q^k$), or at line 26 (in which case $m \in Cg\_Deliver_q^k$).

(b) Since p does not execute Consensus in stage k, by Lemma 4, no correct process executes Consensus in stage k. Therefore, $m \in localNCg\_Deliver_p^k$, and it must be that p has received n_{ack} messages of the type $(k, pending^k, ACK)$ (line 30) such that $m \in pending^k$. There are $n_{ack} \geq (n+1)/2$ processes correct, and so, p has received the message $(k, pending^k, ACK)$ from at least one correct process r.

We claim that every correct process r' executes the $send(k, pending^k, ACK)$ statement at line 15, such that $m \in pending^k$. From lines 12-15, r R-delivers m, and by the agreement of Reliable Broadcast, eventually r' also R-delivers m. It follows from the fact that m is g-Delivered by p in stage k that $m \notin \cup_{i=1}^{k-1}(localNCg\_Deliver_p^k \cup NCg\_Deliver_p^k \cup Cg\_Deliver_p^k)$. By Lemma 4, $m \notin \cup_{i=1}^{k-1}(localNCg\_Deliver_{r'}^k \cup NCg\_Deliver_{r'}^k \cup Cg\_Deliver_{r'}^k)$. Thus, there is a time after r' R-delivers m such that $m \notin \cup_{i=1}^{k-1}(localNCg\_Deliver_{r'}^k \cup NCg\_Deliver_{r'}^k \cup Cg\_Deliver_{r'}^k)$, and m does not conflict with any other message, and so, r' executes send$(k, -, ACK)$ at line 15, concluding the claim.

Since there are n_{ack} processes correct that from the claim above execute the $send(k, pending^k, ACK)$ statement at line 15, such that $m \in pending^k$, q will eventually execute the *when* statement at line 31, and g-Deliver m. $\Box$

Proposition 2 (Partial Order). *If correct processes p and q both g-Deliver messages m and m', and m and m' conflict, then p g-Delivers m before m' if and only if q g-Delivers m before m'.*

PROOF: Assume that messages m and m' conflict, and that q g-Delivers message m before message m'. The following cases cover all combinations involving the g-Deliver of m and m' by q:

(a) m and m' are g-Delivered by q in stage k, and q executes Consensus in stage k,

(b) m and m' are g-Delivered by q in stage k, and q does not execute Consensus in stage k, and

(c) m is g-Delivered by q in stage k, and m' is g-Delivered by q in stage $k', k \neq k'$.

Case (a). It follows that m and m' are in $localNCg\_Deliver_q^k \cup NCg\_Deliver_q^k \cup Cg\_Deliver_q^k$. We claim that $m' \in Cg\_Deliver_q^k$. For a contradiction, assume that $m' \notin Cg\_Deliver_q^k$. Thus, either (a.1) $m' \in localNCmsgSet_q^k$, or (a.2) $m' \in NCg\_Deliver_q^k$.

(a.1) If $m' \in localNCmsgSet_q^k$, then q received n_{ack} messages of the type $(k, pending^k, ACK)$, such that $m' \in pending^k$. By the hypothesis, m is g-Delivered before m', and this can only happen if $m \in localNCmsgSet_q^k$. Thus, q also received n_{ack} messages of the type $(k, pending^k, ACK)$ such that $m \in pending^k$. Since $n_{ack} > n/2$, there must be at least one process r that executed $send(k, pending_r^k, ACK)$, such that m and m' are in $pending_r^k$, contradicting Lemma 3.

(a.2) Since $m' \in NCmsgSet^k$, from validity of Consensus, there is a process r that executed $propose(k, (majMSet_r^k, -))$, such that $NCmsgSet^k = majMSet_r^k$. Therefore, $m' \in majMSet_r^k$, and from the algorithm, r received $\lceil (n_{chk} + 1)/2 \rceil$ messages of the type $(k, pending^k)$ such that $m' \in pending^k$. Since m is g-Delivered before m', either (i) $m \in localNCmsgSet_q^k$ and, from the algorithm, m is g-Delivered before the k-th Consensus execution, or (ii) $m \in NCg\_Deliver_q^k$ and q chooses to g-Deliver m before m' (line 25). We proceed by showing that in both cases, there is at least one process r' such that m and m' are in $pending_{r'}^k$. Since m and m' conflict, this contradicts Lemma 3.

(i) Process q received n_{ack} messages of the type $(k, pending^k, ACK), m \in pending^k$. Let $n_{maj} = |chkPSet^k(m)|$. From line 20, $n_{maj} \geq (n_{chk} + 1)/2$. Since $n_{chk} \geq 2(n - n_{ack}) + 1$, $n_{maj} \geq n - n_{ack} + 1$, thus, $n_{ack} + n_{maj} \geq n + 1$, and we conclude that there is at least on process r' such that m and m' are in $pending_{r'}^k$.

(ii) It must be that $m \in majMSet_r^k$, and thus, r received $\lceil (n_{chk} + 1)/2 \rceil$ messages of the type $(k, pending^k)$ such that $m' \in pending^k$. From item (i), $n_{maj} \geq n - n_{ack} + 1$, and from the hypothesis, $n_{ack} \geq (n + 1)/2$. Therefore, $n_{maj} \geq (n + 1)/2$, and there must exist an r' such that m and m' are in $pending_{r'}^k$, concluding our claim.

If $m \in localNCg\_Deliver^k_q \cup NCg\_Deliver^k_q$, by Lemma 4, $m \in localNCg\_Deliver^k_p \cup NCg\_Deliver^k_p$. From the claim, $m \in Cg\_Deliver^k_q$, and so, by Lemma 4, $m \in Cg\_Deliver^k_p$. It follows from the algorithm that messages in $localNCg\_Deliver^k \cup NCg\_Deliver^k$ are g-Delivered before messages in $Cg\_Deliver^k$. If m and m' are in $Cg\_Deliver^k_q$, they are also in $Cg\_Deliver^k_p$ ($Cg\_Deliver^k_q = Cg\_Deliver^k_p$), and since messages in $Cg\_Deliver^k$ are g-Delivered according to some deterministic order, if q g-Delivers m before m', p g-Delivers m before m'.

Case (b). Since there is no Consensus execution in stage k, m and m' are in $localNCg\_Deliver^k_q$. It follows from the same argument used in (a.1) that there must be a process r such that m and m' are in $pending^k_r$, and this contradicts Lemma 3. Thus, it cannot be that m and m' are g-Delivered by q in stage k if q does not execute Consensus in stage k.

Case (c). If q g-Delivers m before m', then $k < k'$. It follows immediately from Lemma 4 that p does not g-Deliver m' before m. □

Proposition 3 (Validity). *Let $f < max(n_{ack}, n_{chk})$. If a correct process p g-Broadcasts a message m, then p eventually g-Delivers m.*

Proof: For a contradiction, assume that p g-Broadcasts m but never g-Delivers it. From Lemma 1, no correct process g-Delivers m. Since p g-Broadcasts m, it R-broadcasts m, and from the validity of Reliable Broadcast, p eventually R-delivers m. From the agreement of Reliable Broadcast, there is a time after which for every correct process q, $m \in (R\_delivered_q \setminus G\_delivered_q) \setminus pending^k_q$.

By the hypothesis, p does not g-Deliver m, and so, p does not receive n_{ack} messages of the type $(k, pending^k, ACK)$ such that $m \in pending^k$. But since there are n_{ack} processes correct that execute the *if* statement at line 13, there is at least one correct process q that never executes the *then* branch (lines 14-15), that is, $send(k, pending^k_q, ACK)$, and always executes the *else* branch (lines 17-30). Thus, q executes $send(k, pending^k, CHK)$. From Lemma 4, part (1), every correct process also executes $send(k, pending^k, CHK)$. Since there are n_{chk} processes correct, no correct process remains blocked forever at the *wait* statement (line 18), and every correct process executes $propose(k, -)$. Thus, there is a k_1 such that for all $l \geq k_1$, all correct processes execute $propose(l, (majMSet^l, (R\_delivered \setminus G\_delivered) \setminus majMSet^l))$ such that $m \in majMSet^l \cup (R\_delivered \setminus G\_delivered)$.

Since all faulty processes crash, there is a k_2 such that no faulty process executes $propose(l, -)$, $l \geq k_2$. Let $k = max(k_1, k_2)$. All correct processes execute $propose(k, -)$, and by the termination and agreement of consensus, all correct processes execute $decide(k, (NCmsgSet^k, CmsgSet^k))$ with the same $(NCmsgSet^k, CmsgSet^k)$. By uniform validity of Consensus, some process

q executes $propose(k, (majMSet^l, (R\_delivered \setminus G\_delivered) \setminus majMSet^l))$ such that $m \in majMSet^l \cup (R\_delivered \setminus G\_delivered)$, and so, all processes g-Deliver m, a contradiction that concludes the proof. □

Proposition 4 (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: If a process p g-Delivers m at line 36, then p received n_{ack} messages of the type $(k, pending^k, ACK), m \in pending^k$. Let q be a process from which p received the message $(k, pending_q^k, ACK), m \in pending_q^k$. Since q executes $send(k, pending_q^k, ACK)$, q has R-delivered m. By the uniform integrity of Reliable Broadcast, process *sender(m)* R-broadcast m, and so, *sender(m)* g-Broadcast m.

Now consider that p g-Delivers m at line 25 or 26. Thus, p executed $decide(k, (NCmsgSet^k, CmsgSet^k))$ for some k, and such that $m \in NCmsgSet^k \cup CmsgSet^k$. By uniform validity of Consensus, some process q must have executed $propose(k, (majMSet^k, (R\_delivered \setminus G\_delivered) \setminus majMSet^k))$ such that $m \in majMSet^k \cup (R\_delivered \setminus G\_delivered)$. If $m \in majMSet_q^k$, then $chkPSet_q^k(m) \neq \emptyset$. Let $r \in chkPSet_q^k(m)$. Thus, r executed $send(k, pending_r^k, CHK)$, such that $m \in pending_r^k$, and thus, r R-delivered m. If $m \in R\_delivered \setminus G\_delivered$ then q has R-delivered m. In any case, by the uniform integrity of Reliable Broadcast, process *sender(m)* R-broadcast m, and so, *sender(m)* g-Broadcast m. □

Theorem 1 *If $f < max(n_{ack}, n_{chk})$, Algorithm 1 solves Generic Broadcast, or reduces Generic Broadcast to a sequence of Consensus problems.*

PROOF. Immediate from Propositions 1, 2, 3, and 4. □

Proposition 5 *Algorithm 1 is a strict Generic Broadcast algorithm.*

PROOF. From Theorem 1, Algorithm 1 is a Generic Broadcast algorithm. We show next that it is also strict. Let $\mathcal{R}^{NC}$ be the set of runs generated by Algorithm 1 in which no conflicting messages are g-Broadcast. We show that there is some run R in $\mathcal{R}^{NC}$ and messages m' and m'' g-Broadcast in R such that some process r' g-Delivers m' before m'' in R, and some process r'' g-Delivers m'' before m' in R.

We construct run R as follows. Assume that r' and r'' are two process correct in R, and let $m \in \{m', m''\}$. If m is g-Broadcast, then m is R-broadcast. By the agreement and validity of Reliable Broadcast, eventually every correct process R-delivers m. Since no message conflicts in R, after a process p R-delivers m, p executes $send(1, pending^1, ACK)$, such that $m \in pending^1$. Assume that when p executes $send(1, pending^1, ACK)$, m' and m'' are in $pending^k$. Since there are n_{ack} process correct, eventually every process correct receives n_{ack} messages of the type

$(1, \mathit{pending}^1, ACK)$, such that, m' and m'' are in $\mathit{pending}^1$. Therefore, r' and r'' eventually g-Deliver messages m' and m'' in $ackMSet^1$. If r' g-Delivers m' before m'' than assume that r'' g-Delivers m'' before m', and if r' g-Delivers m'' before m' than assume that r'' g-Delivers m' before m''. □

Proposition 6 *Assume that Algorithm 1 uses the Reliable Broadcast implementation presented in [2]. If $\mathcal{R_C}$ is a set of runs generated by Algorithm 1 such that m is the only message g-Broadcast and g-Delivered in runs in $\mathcal{R_C}$, then there is no run R in $\mathcal{R_C}$ where $dl^R(m) < 2$.*

PROOF. Assume for a contradiction that there is a run R in $\mathcal{R_C}$ such that $dl^R(m) < 2$. Let p be a correct process that g-Delivers m in R. By the integrity of Generic Broadcast, there is a process q that g-Broadcast m. From Algorithm 1, q R-broadcast m, and by the implementation of Reliable Broadcast, q sends m to all processes. Let $ts(\mathit{send}_q(m))$ be the timestamp of the send event at q. When p receives m, we have that $ts(\mathit{receive}_p(m)) = ts(\mathit{send}_q(m)) + 1$. From the contradiction hypothesis and the definition of deliver latency, $\textit{g-Deliver}_p(m) - \textit{g-Broadcast}_q(m) < 2$, and so, after receiving m, p does not receive any message m' such that $m \rightarrow m'$. There are two cases to consider: either (a) p g-Delivers m at line 36, or (b) p g-Delivers m in lines 25-26. In case (a), p receives n_{ack} messages of the type $(k, \mathit{pending}^k, ACK)$, such that $m \in \mathit{pending}^k$. Let r be a process that sends message $(k, \mathit{pending}^k, ACK)_r$ at line 17. If $m \in \mathit{pending}^k$, than r has received m, and so, $m \rightarrow (k, \mathit{pending}^k, ACK)_r$, a contradiction. In case (b), p has received n_{chk} messages of the type $(k, \mathit{pending}^k, CHK)$, such that in $(n_{chk} + 1)/2$ messages, $m \in \mathit{pending}^k$. Let r be a process that sends message $(k, \mathit{pending}^k, CHK)_r$ such that $m \in \mathit{pending}^k$. It can be show that $m \rightarrow (k, \mathit{pending}^k, CHK)_r$, contradicting the fact that p does not receive any message m' such that $m \rightarrow m'$. □

Proposition 7 *Assume that Algorithm 1 uses the Reliable Broadcast implementation presented in [2]. If $\mathcal{R_C}$ is a set of runs generated by Algorithm 1, such that in runs in $\mathcal{R_C}$, m is the only message g-Broadcast and g-Delivered, and there are no process failures nor failure suspicions, then there is a run R in $\mathcal{R_C}$ where $dl^R(m) = 2$.*

PROOF. Immediate from Algorithm 1. □

Proposition 8 *Assume that Algorithm 1 uses the Reliable Broadcast implementation presented in [2], and the Consensus implementation presented in [10]). Let $\mathcal{R_C}$ be a set of runs generated by Algorithm 1, such that m and m' are the only messages g-Broadcast and g-Delivered in $\mathcal{R_C}$. If m and m' conflict, then there is no run R in $\mathcal{R_C}$ where $dl^R(m) < 4$ and $dl^R(m') < 4$.*

PROOF. Assume for a contradiction that there is a run R in $\mathcal{R_C}$ such that $dl^R(m) < 4$ and $dl^R(m') < 4$. Let p be a correct process that g-Delivers m and m' in R. By the integrity of Generic Broadcast, there are processes q and q' that g-Broadcast m and m', respectively. From Algorithm 1, q and q' R-broadcast m and m', and by the implementation of Reliable Broadcast, q and q' send m and m' to all processes. Without loss of generality, consider that p first receives m and then m'. We will reach a contradiction by showing that $dl^R(m') \geq 4$. Let $ts(send_{q'}(m'))$ be the timestamp of the send event at q'. When p receives m', we have that $ts(receive_p(m')) = ts(send_{q'}(m')) + 1$.

After receiving m', there is a time t when p executes the *when* statement at line 12 such that $m' \in R\_delivered_p \setminus G\_delivered_p$, and $m' \notin pending_p^k$. Since m is received by p before m', at time t both m and m' are in $R\_delivered_p \setminus G\_delivered_p$. Since m and m' conflict, p executes the *else* branch of the *if* statement at line 13, and at line 18 p receives n_{chk} messages of the type $(k, pending^k, CHK)$, such that in $(n_{chk} + 1)/2$ messages, $m' \in pending^k$. Let r be a process that sends message $(k, pending^k, CHK)_r$ such that $m' \in pending^k$. It follows that $m' \rightarrow (k, pending^k, CHK)_r$, and so, $ts(receive_p(k, pending^k, CHK)) = ts(send_{q'}(m')) + 2$. From the contradiction hypothesis and the definition of deliver latency, $ts(g\text{-}Deliver_p(m)) - ts(g\text{-}Broadcast_q(m)) < 4$. Since $ts(g\text{-}Broadcast_q(m)) = ts(send_{q'}(m'))$, and $ts(g\text{-}Deliver_p(m)) = ts(send_{q'}(m')) + 2 + C$, where C is the length of the causal chain of messages generated by the Consensus execution, we conclude that $C < 2$. This leads to a contradiction since for the Consensus algorithm presented in [10], the minimal causal chain of messages is 2, and therefore, $C \geq 2$. □

Proposition 9 *Assume that Algorithm 1 uses the Reliable Broadcast implementation presented in [2], and the Consensus implementation presented in [10]). Let $\mathcal{R_C}$ be a set of runs generated by Algorithm 1, such that m and m' are the only messages g-Broadcast and g-Delivered in $\mathcal{R_C}$, and there are no process failures nor failure suspicions. If m and m' conflict, then there is a run R in $\mathcal{R_C}$ where m is g-Delivered before m' and $dl^R(m) = 2$ and $dl^R(m') = 4$.*

Proof. Immediate from Algorithm 1. □