# Generic Broadcast*

Fernando Pedone and André Schiper

Département d'Informatique
Ecole Polytechnique Fédérale de Lausanne
CH-1015 Lausanne EPFL, 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 *delivery 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 [12]), 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

* Research supported by the EPFL-ETHZ DRAGON project and OFES under contract number 95.0830, as part of the ESPRIT BROADCAST-WG (number 22455).

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), by 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.

In order to assess the cost of Generic Broadcast algorithms, we introduce the concept of *delivery latency* of a message. Roughly speaking, the delivery 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 is less expensive than known Atomic Broadcast algorithms, that is, in runs where messages do not conflict, our algorithm ensures that the delivery latency of every message is always equal to 2 (known Atomic Broadcast algorithms have at least delivery 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 delivery 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

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

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:

**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, then $p$ 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 $\mathcal{C}_{Account}$:

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

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 $\mathcal{R}_{\mathcal{C}}^{NC}$ be the set of runs of $A_{\mathcal{C}}$. There exists a run $R$ in $\mathcal{R}_{\mathcal{C}}^{NC}$, in which at least two correct processes g-Deliver two non-conflicting 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] 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 $x$.

## 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 the maximum number of failures is smaller than $n/2$) [2].

### 3.2 Delivery Latency

In the following, we introduce the delivery latency as a parameter to measure the efficiency of algorithms solving a Broadcast problem (defined by the primitives $\alpha$-Broadcast and $\alpha$-Deliver). The deliver latency is a variation of the Latency Degree introduced in [11], which is based on modified Lamport's clocks [7].

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

The delivery latency of a message $m$ $\alpha$-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 $\alpha$-Deliver($m$) events (at most one per process) in run $R$, and the timestamp of the $\alpha$-Broadcast($m$) event in run $R$.

Let $\pi^R(m)$ be the set of processes that $\alpha$-Deliver message $m$ in run $R$, and $\alpha\text{-Deliver}_p(m)$ the $\alpha$-Deliver($m$) event at process $p$. The deliver latency of $m$ in run $R$ is formally defined as

$$dl^R(m) \stackrel{def}{=} \underset{p \in \pi^R(m)}{MAX} \left(ts(\alpha\text{-Deliver}_p(m)) - ts(\alpha\text{-Broadcast}(m))\right).$$

For example, consider a broadcast algorithm where a process $p$, wishing to broadcast a message $m$, (1) sends $m$ to all processes, (2) each

process $q$ on receiving $m$ sends an acknowledge message $ACK(m)$ to all processes, and (3) as soon as $q$ receives $n_{ack}$ messages of the type $ACK(m)$, $q$ delivers $m$. Let $R$ be a run of this algorithm where only $m$ is broadcast. We have $dl^R(m) = 2$.

## 4 Solving Generic Broadcast

### 4.1 Overview of the Algorithm

Processes executing our Generic Broadcast algorithm progress in a sequence of stages numbered $1, 2, ..., k, ....$ Stage $k$ terminates only if two conflicting messages are g-Broadcast, but not g-Delivered in some stage $k' < k$.

*g-Delivery of non-conflicting messages.* Let $m$ be a message that is g-Broadcast. When some process $p$ receives $m$ in stage $k$, and $m$ does not conflict with some other message $m'$ already received by $p$ in stage $k$, then $p$ inserts $m$ in its $pending_p^k$ set, and sends an *ACK(m)* message to all processes. As soon as $p$ receives *ACK(m)* messages from $n_{ack}$ processes, where

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

$p$ g-Delivers $m$.

*g-Delivery of conflicting messages.* If a conflict is detected, Consensus is launched to terminate stage $k$. The Consensus decides 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 all messages that are g-Delivered in stage $k$. Messages in $NCmsgSet^k$ are g-Delivered before messages in $CmsgSet^k$, and messages in $NCmsgSet^k$ may be g-Delivered by some process $p$ in stage $k$ before $p$ executes the $k$-th Consensus. 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 $CmsgSet^k$.

*Properties.* To be correct, our algorithm must satisfy the following properties:

(a) If two messages $m$ and $m'$ conflict, then at most one of them is g-Delivered in stage $k$ before Consensus.

(b) If message $m$ is g-Delivered in stage $k$ by some process $p$ before Consensus, then $m$ is in the set $NCmsgSet^k$.
(c) The set $NCmsgSet^k$ does not contain any conflicting messages.[3]

Property (a) is ensured by condition (1). Property (b) is ensured as follows. Before starting Consensus, every process $p$ sends its $pending_p^k$ set to all processes (in a message of type *checking*, denoted by CHK), and waits for messages of type CHK from exactly $n_{chk}$ processes. Only if some message $m$ is at least in $\lceil (n_{chk}+1)/2 \rceil$ messages of type CHK, then $m$ is inserted in $majMSet_p^k$, the initial value of Consensus that decides on $NCmsgSet^k$. So, if $m$ is in less than $\lceil (n_{chk}+1)/2 \rceil$ messages of type CHK, $m$ is not inserted in $majMSet_p^k$. Indeed, if condition

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

holds, then $m$ could not have been g-Delivered in stage $k$ before Consensus. To understand why, notice that from (2), we have

$$(n - n_{chk}) + \lceil (n_{chk}+1)/2 \rceil \leq n_{ack}, \tag{3}$$

where $(n - n_{chk})$ is the number of processes from which $p$ knows nothing. From (3), if $m$ is in less than $\lceil (n_{chk}+1)/2 \rceil$ messages of type CHK, then even if all processes from which $p$ knows nothing had sent $ACK(m)$, there would not be enough $ACK(m)$ messages to have $m$ g-Delivered by some process in stage $k$ before Consensus.

Property (c) is ensured by the fact that $m$ is inserted in $majMSet_p^k$ only if $m$ is in at least $\lceil (n_{chk}+1)/2 \rceil$ messages of type CHK received by $p$ (majority condition). Let $m$ and $m'$ be two messages in $majMSet_p^k$. By the majority condition, the two messages are in the $pending_q^k$ set of at least one process $q$. This is however only possible if $m$ and $m'$ do not conflict.

*Minimal number of correct processes.* Our Generic Broadcast algorithm waits for $n_{ack}$ messages before g-Delivering non-conflicting messages, and $n_{chk}$ messages if a conflict is detected before starting Consensus. So our algorithm requires $max(n_{ack}, n_{chk})$ correct processes. The minimum of this expression happens to be $(2n+1)/3$, when $n_{ack} = n_{chk}$.

[3] Property (c) does not follow from (a) and (b). Take for example two messages $m$ and $m'$ that conflict, but are not g-Delivered in stage $k$ without the cost of Consensus: neither property (a), nor property (b) applies.

### 4.2 The Generic Broadcast Algorithm

Provided that the number of correct processes is at least $max(n_{ack}, n_{chk})$, $n_{ack} \geq (n+1)/2$, and $2n_{ack} + n_{chk} \geq 2n + 1$, Algorithm 1 solves Generic Broadcast for any conflict relation $\mathcal{C}$. 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 exactly $n_{chk}$ 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] $majMSet_p^k = \{m : |Chk_p^k(m)| \geq (n_{chk}+1)/2\}$

**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 $(\underline{k}, pending_q^k, \underline{CHK})$ from $q$ ]
19: $\#Define\ Chk^k(m) = \{q : p$ received $(\underline{k}, pending_q^k, \underline{CHK})$ from $q$ **and** $m \in pending_q^k\}$
20: $majMSet^k \leftarrow \{m : |\, Chk^k(m)\,| \geq \lceil (n_{chk} + 1)/2 \rceil\}$
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$(\underline{k}, pending_q^k, \underline{ACK})$ from $q$
32: $\#Define\ Ack^k(m) = \{q : p$ received $(\underline{k}, pending_q^k, \underline{ACK})$ from $q$ **and** $m \in pending_q^k\}$
33: $ackMSet^k \leftarrow \{m : |Ack^k(m)| \geq n_{ack}\}$
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

### 4.3 Proof of Correctness

Due to space limitations, we have only included some of the proofs in this section. All proofs (Agreement, Partial Order, Validity, and Integrity) can be found in [9]. In the following, we prove that the three properties ((a)-(c)) presented in Section 4.1 hold.

Lemma 1 states that the set $pending^k$ does not contain conflicting messages. It is used to prove Lemmata 2 and 5 below.

**Lemma 1.** *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: Suppose, by way of contradiction, that there is a process $p$, and some $k \geq 1$ such that $m$ and $m'$ conflict and are in $pending_p^k$. Since $m$ and $m'$ are in $pending_p^k$, $p$ must have R-delivered $m$ and $m'$. Assume that $p$ first R-delivers $m$ and then $m'$. Thus, there is a time $t$ 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 $m' \notin pending_p^k$. At time $t$, $m \in R\_delivered_p$ (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, when the *if* statement at line 13 is evaluated, $m$ and $m'$ are in $R\_delivered \setminus G\_delivered$, and since $m$ and $m'$ conflict, the condition evaluates false, and $m'$ is not included in $pending_p^k$, a contradiction that concludes the proof. □

Lemma 2 proves property (a).

**Lemma 2.** *If two messages $m$ and $m'$ conflict, then at most one of them is g-Delivered in stage $k$ before Consensus.*

PROOF: The proof is by contradiction. Assume that there are two messages $m$ and $m'$ that conflict and are g-Delivered in stage $k$ before Consensus. Without lack of generality, consider that $m$ is g-Delivered by process $p$, and $m'$ is g-Delivered by process $q$. From the Generic Broadcast algorithm (lines 31-36), $p$ ($q$) has received $n_{ack}$ messages of the type $(k, pending^k, ACK)$ such that $m \in pending^k$ ($m' \in pending^k$). Since $n_{ack} > (n+1)/2$, there must be a process $r$ that sends the message $(k, pending_r^k, ACK)$ to processes $p$ and $q$, such that $m$ and $m'$ are in $pending_r^k$, contradicting Lemma 1. □

Lemma 3 relates (1) the set $Ack^k(m)$ of processes that send an acknowledgement for some message $m$ in stage $k$ and (2) the set $Chk_p^k$ of

processes from which some process $p$ receives $CHK$ messages in stage $k$, with (3) the set $Chk_p^k(m)$ of processes from which $p$ receives a $CHK$ message containing $m$ in stage $k$.

**Lemma 3.** *Let $Ack^k(m)$ be a set of processes that execute the statement $send(k, pending^k, ACK)$ (line 15) in stage $k$ with $m \in pending^k$, and let $Chk_p^k$ be the set of processes from which some process $p$ receives messages of the type $(k, pending^k, CHK)$ in stage $k$ (line 18). If $Ack^k(m) \geq n_{ack}, Chk_p^k \geq n_{chk}$, and $2n_{ack} + n_{chk} \geq 2n + 1$, then there are at least $\lceil (n_{chk} + 1)/2 \rceil$ processes in $Chk_p^k(m) \stackrel{def}{=} Chk_p^k \cap Ack^k(m)$.*

PROOF: We prove the contrapositive, that is, if $|Chk_p^k(m)| < \lceil (n_{chk} + 1)/2 \rceil$ then $|Ack^k(m)| < n_{ack}$. From the definitions of $Ack^k(m)$ and $Chk_p^k(m)$, it follows that $|Ack^k(m)| \leq (n - n_{chk}) + |Chk_p^k(m)|$ (1). To see why, notice that set $Chk_p^k(m)$ contains all processes from set $Chk_p^k$ that sent an acknowledgement message for $m$. Process $p$ does not know anything about the remaining processes in $\Pi \setminus Chk_p^k$, but even if all of them acknowledged message $m$, the number of acknowledges is at most equal to $(n - n_{chk})$.

From (1) and the fact that $|Chk_p^k(m)| < \lceil (n_{chk} + 1)/2 \rceil$, we have $|Ack^k(m)| - (n - n_{chk}) \leq |Chk_p^k(m)| < \lceil (n_{chk} + 1)/2 \rceil$. Thus, $(n - n_{chk}) > |Ack^k(m)| - \lceil (n_{chk} + 1)/2 \rceil$ (2). From $2n_{ack} + n_{chk} \geq 2n + 1$, and the fact that $n_{ack}$, $n_{chk}$, and $n$ are integers, we have that $(n - n_{chk}) \leq n_{ack} - \lceil (n_{chk} + 1)/2 \rceil$ (3). Therefore, from (2) and (3), we conclude that $|Ack^k(m)| < n_{ack}$. □

Lemma 4 proves property (b) presented in Section 4.1. It 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 4.** *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 3, $Chk^k(m) \geq (n_{chk} + 1)/2$, and so, from the algorithm line 20, $m \in majMSet_p^k$. Therefore, for every process $q$ that exe-

cutes $propose(k, (majMSet_q^k, -))$, $m \in majMSet_q^k$. 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 5 proves property (c).

**Lemma 5.** *If two messages $m$ and $m'$ conflict, then at most one of them is in $NCmsgSet^k$.*

PROOF: The proof is by contradiction. Assume that there are two messages $m$ and $m'$ that conflict, and are both in $NCmsgSet^k$. From the validity property of Consensus, there must be a process $p$ that executes $propose(k, (majMSet_p^k, -))$, such that $NCmsgSet^k = majMSet_p^k$. Therefore, $m$ and $m'$ are in $majMSet_p^k$, and from the algorithm, $p$ receives $\lceil (n_{chk}+1)/2 \rceil$ messages of the type $(k, pending^k, CHK)$ such that $m$ is in $pending^k$, and $p$ also receives $\lceil (n_{chk}+1)/2 \rceil$ messages of the type $(k, pending^k, CHK)$ such that $m'$ is in $pending^k$. Since $p$ waits for $n_{chk}$ messages of the type $(k, pending^k, CHK)$, there must exist at least one process $q$ in $Chk_p^k$ such that $m$ and $m'$ are in $pending_q^k$, contradicting Lemma 1. □

### 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 delivery latency equal to 2, while for messages that conflict, the delivery latency is at least equal to 4. Since known Atomic Broadcast algorithms deliver messages with a delivery 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.

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

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 delivery 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 delivery latency of the algorithm, and Proposition 7 shows that this bound can be reached in runs where there are no process failures. 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, 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 [11]. 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 [11]). 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 $dl^R(m') = 4$.*

[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 $q$ R-delivers $m$.

## 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 increase the performance of the system. Lazy replication [10] is another approach that aims at increasing the performance 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 performance of replication by increasing concurrency in the context of group communication. 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

[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 $n_r + n_w \geq n + 1$.

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 delivery 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.

## 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. F. Pedone and A. Schiper. Generic broadcast. Technical Report SSC/1999/012, EPFL, Communication Systems Department, April 1999.
10. S. Ghemawat R. Ladin, B. Liskov. Providing High Availability Using Lazy Replication. *ACM Transactions on Computer Systems*, 10(4):360–391, November 1992.
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.