Thrifty Generic Broadcast*

Marcos Kawazoe Aguilera[1], Carole Delporte-Gallet[2], Hugues Fauconnier[2], and Sam Toueg[1]

[1] DIX, École Polytechnique, 91128 Palaiseau Cedex, France,
aguilera,sam@lix.polytechnique.fr
[2] LIAFA, Université D. Diderot, 2 Place Jussieu, 75251 Paris Cedex 05, France,
cd,hf@liafa.jussieu.fr

Abstract. We consider the problem of generic broadcast in asynchronous systems with crashes, a problem that was first studied in [12]. Roughly speaking, given a “conflict” relation on the set of messages, generic broadcast ensures that any two messages that conflict are delivered in the same order; messages that do not conflict may be delivered in different order. In this paper, we define what it means for an implementation generic broadcast to be “thrifty”, and give corresponding implementations that are optimal in terms of resiliency. We also give an interesting application of our results regarding the implementation of atomic broadcast.

1 Introduction

Atomic broadcast is a well-known building block of fault-tolerant distributed applications (e.g., see [7, 4, 9, 8, 10, 3, 2]). Informally, this communication primitive ensures that *all* messages broadcast are delivered in the same order. In a recent paper, Pedone and Schiper noted that for some applications some messages do not “conflict” with each other, and hence they can be delivered by different processes in different orders [12]. For such applications, the broadcast communication primitive does not need to order all messages; it must order only the conflicting ones. An example given in [12] consists of *read* and *write* messages broadcast to replicated servers, where read messages do not conflict with each other, and hence do not have to be ordered. Intuitively, one may want to avoid ordering the delivery of messages unless it is really necessary: such ordering may be expensive, or even impossible unless one uses oracles such as as failure detectors, and these can be unreliable.

In view of the above, Pedone and Schiper proposed a generalized version of atomic broadcast, called *generic broadcast*. Informally, given any *conflict* relation defined over the set of messages, *if* two messages m and m' conflict, then generic broadcast ensures that they are delivered in the same order.[1] Messages that do not conflict are not required to be ordered. Note that if the conflict relation includes all the pairs of messages, generic broadcast coincides with atomic

* Research partially supported by NSF grants CCR-9711403.

[1] The conflict relation is a parameter of generic broadcast. We assume that it is symmetric and non-reflexive.

broadcast. On the other hand, if the conflict relation is empty, generic broadcast reduces to reliable broadcast.

How can one implement a generic broadcast primitive? A trivial way is to use atomic broadcast to broadcast *every* message that we want to gbroadcast.[2] This ensures that all messages are ordered, including non-conflicting ones. Such an implementation is unsatisfactory, and goes against the motivation for introducing generic broadcast in the first place. To avoid this trivial implementation, and in order to characterize "good" implementations, Pedone and Schiper introduced the notion of *strictness*. Roughly speaking, an implementation of generic broadcast is strict if it has at least one execution in which two processes deliver two non-conflicting messages in a different order. The notion of strictness is intended to capture the intuitive idea that the total order delivery of messages has a cost, and this cost should be paid only when necessary. As Pedone and Schiper point out in [13], however, the strictness requirement is not sufficient to characterize good implementations of generic broadcast. Intuitively, this is because there is a strict implementation that first orders *all* the messages, including non-conflicting ones, and then selects two non-conflicting messages and delivers them in different orders. Even though such an implementation is strict, it goes against the motivation behind generic broadcast.

In this paper, we reconsider the question of what it means for an implementation of generic broadcast to be good, and we propose new definitions. We first note that in asynchronous systems with crash failures (the systems considered in [12] and here), generic broadcast cannot be implemented without the help of an "oracle" that can be used to order the delivery of messages that conflict. This oracle could be a "box" that solves atomic broadcast or consensus; or it could be a failure detector that can be used to implement such a box. In the first case, this oracle is expensive; in the second case, it can be unreliable and its mistakes can slow down the delivery of messages.[3] In either case, one should avoid the use of the oracle whenever possible. Thus, a good implementation of generic broadcast is one that takes advantage of the fact that only conflicting messages need to be ordered, and uses its oracle only when there are conflicting messages that are actually broadcast.

This leads us to the following definition. Roughly speaking, an implementation of generic broadcast is *non-trivial w.r.t. an oracle*, if it satisfies the following property: if all the messages that are actually broadcast do not conflict with each other, then the oracle is never used. A non-trivial implementation, however, is still unsatisfactory: even in a run where there is *only one* broadcast that conflicts with a previous one, such an implementation is allowed use its oracle an

[2] Henceforth, *gbroadcast* and *gdeliver* are the two primitives associated with generic broadcast. Similarly, *abroadcast* and *adeliver* are associated with atomic broadcast.

[3] Even though one can implement failure detectors that are fairly accurate in practice [14, 6], they may have "bad" periods of time when they make too many mistakes to be useful. For example, from [5] there is an atomic broadcast algorithm that never deliver messages out of order, but message delivery is delayed if/when the algorithm happens to rely on the failure detector during one of its bad periods.

unlimited number of times. This motivates our second definition. An implementation of generic broadcast is *thrifty w.r.t. an oracle* if it is non-trivial and it also satisfies the following property: if there is a time after which the messages broadcast do not conflict with each other, then there is a time after which the oracle is not used. It is easy to see that non-trivial implementations and thrifty ones are necessarily strict in the sense of [12].

In this paper, we consider implementations of generic broadcast that use atomic broadcast as the oracle. Atomic broadcast is a natural oracle for the task of totally ordering conflicting messages. Furthermore, any implementation that is thrifty w.r.t. atomic broadcast can be transformed into an implementation that is thrifty w.r.t. consensus. It can also be transformed into an implementation that is thrifty w.r.t. $\diamond\mathcal{S}$, the weakest failure detector that can be used to solve generic broadcast (this last transformation assumes that a majority of processes is correct).

We present two implementations of generic broadcast: one is non-trivial and the other is thrifty. The non-trivial implementation is simple and illustrates some of our basic techniques; the thrifty implementation is more complex and builds upon the simple implementation. Both implementations work for asynchronous systems with n processes where up to $f < n/2$ may crash, which is optimal. Since both implementations are also strict, this improves on the resiliency of the strict implementation given in [12] which tolerates up to $f < n/3$ crashes.

We continue the paper with an interesting use of thrifty implementations of generic broadcast. Specifically, we show how they can be used to derive "sparing" implementations of atomic broadcast, as we now explain. First note that in asynchronous systems with failures, any implementation of atomic broadcast requires the use of an external oracle, and (just as with generic broadcast) it is better to avoid relying on this oracle whenever possible. For example, if the oracle is a failure detector, relying on this oracle during one of its "bad" period can delay the delivery of messages. So we would like an implementation of atomic broadcast that uses the oracle sparingly. How can we do so?

Suppose a process atomically broadcast m and then m'. No oracle is needed to ensure that m and m' are delivered in the same order everywhere: FIFO order can be easily enforced with sequence numbers assigned by the sender. Similarly, suppose two atomic broadcast messages happen to be causally related[4], e.g., m is adelivered by a process before it abroadcasts m'. Then, we can order the delivery of m and m' without any oracle (this can be done with message piggybacking or "vector clocks"; see for example [10]). Thus, an implementation of atomic broadcast can reduce its reliance on the oracle, by restricting its use to the ordering of broadcast messages that are *concurrent*. We say that an implementation of atomic broadcast is *sparing w.r.t. an oracle*, if it satisfies the following property: If there is a time after which the messages broadcast are pairwise causally related, then there is a time after which the oracle is not used.

[4] We say that two messages are causally related or concurrent, if their broadcast events are causally related or concurrent, respectively, in the sense of [11, 10].

We conclude the paper by showing how to transform *any* implementation of atomic broadcast that uses some oracle, into one that is sparing w.r.t the same oracle. To do so, we use a thrifty implementation of generic broadcast and vector clocks.

As a final remark, note that Pedone and Schiper use message latency as a way to evaluate the efficiency of generic broadcast implementations. In "good" runs with no failures and no conflicting messages, their generic broadcast algorithm ensures that every message is delivered within 2δ (assuming δ is the maximum message delay). In this paper, our focus was not on optimizing the latency of messages in these good runs, but rather on reducing the dependency on the oracle whenever possible. These two goals, however, are not incompatible. In fact, we can modify our thrifty implementations of generic broadcast to also achieve a small message latency in good runs. Specifically, we have an implementation that assumes $f < n/3$ and ensures a message delivery within 2δ in such runs (as in [12]). We also have an implementation that works for $f < n/2$ and ensures message delivery within 3δ in good runs. It is worth noting that *even in runs with failures and conflicting messages*, the message delivery times of 2δ and 3δ, respectively, are *eventually* achieved provided there is a time after which the messages broadcast are not conflicting.

In summary, this paper considers the problem of generic broadcast in asynchronous systems with crashes, a problem that was first studied in [12]. We first propose alternative definitions of "good" implementations of generic broadcast (the previous definition in terms of "strictness" had some drawbacks). Roughly speaking, we consider an implementation to be good if it does not rely on any oracle when the messages that are broadcast do not conflict. We then give two such implementations (with atomic broadcast as its oracle): one does not use the oracle in runs where no messages conflict, and the other one stops using the oracle if conflicting broadcasts cease. Both implementations are optimal in terms of resiliency; they tolerate up to $f < n/2$ process crashes (an improvement over [12]). We then use our results to give "sparing" implementations of atomic broadcast, i.e., implementations that stop using their oracle if concurrent broadcasts cease. Finally, we show how to transform any implementation of atomic broadcast into a sparing one.

In this extended abstract, we omit the proofs (they are given in the full paper [1]).

2 Informal Model

We consider *asynchronous* distributed systems. To simplify the presentation of our model, we assume the existence of a discrete global clock. This is merely a fictional device: the processes do not have access to it. We take the range τ of the clock's ticks to be the set of natural numbers $\mathbf{N}$.

The system consists of a set of n *processes*, $\Pi = \{1, 2, \ldots, n\}$ and an oracle. Processes are connected with each other through reliable asynchronous communication channels. Up to f processes can fail by *crashing*. A failure pattern indicates which processes crash, and when, during an execution. Formally, a *failure pattern* F is a function from τ to 2^{Π}, where $F(t)$ denotes the set of processes that have crashed through time t. Once a process crashes, it does not "recover", i.e., $\forall t : F(t) \subseteq F(t+1)$. We define $\mathit{crashed}(F) = \bigcup_{t \in \tau} F(t)$ and $\mathit{correct}(F) = \Pi - \mathit{crashed}(F)$. If $p \in \mathit{crashed}(F)$ we say *p crashes (in F)* and if $p \in \mathit{correct}(F)$ we say *p is correct (in F)*.

A distributed algorithm $\mathcal{A}$ is a collection of n deterministic automata (one for each process in the system). The execution of $\mathcal{A}$ occurs in *steps* as follows. For every time $t \in \tau$, at most one process takes a step; moreover, every correct process takes an infinite number of steps. In each step, a process (1) may send a message to a process; (2) queries the oracle (the query may be $\perp$); (3) receives an answer from the oracle (possibly $\perp$); (4) receives a message (possibly $\perp$); and (5) changes state. We say that *a process uses the oracle at time t* if it performs a non-$\perp$ query at time t.

An oracle history H is a sequence of quadruples (p, t, i, o), where p is a process, t is a time (t is monotonically increasing in H), i is the query of p at time t, and o is the answer of the oracle to p at time t. We assume that if no process ever uses the oracle (all queries in H are $\perp$) then the oracle never gives any answer (all answers in H are $\perp$). An oracle $\mathcal{O}$ is function that takes a failure pattern F and returns a set $\mathcal{O}(F)$ of oracle histories[5]. Oracles of interest include failure detectors [5], an atomic broadcast black-box, and a consensus black-box. For example, an atomic broadcast black-box can be modeled as an oracle that accepts "broadcast(m)" queries, and outputs "deliver(m)" answers, where the queries/answers satisfy the usual specification of atomic broadcast (see Section 2.2).

2.1 Reliable broadcast

Intuitively, reliable broadcast ensures that processes agree on the set of messages that they deliver. More precisely, *reliable broadcast* is defined in terms of two primitives: $\mathit{rbroadcast}(m)$ and $\mathit{rdeliver}(m)$. We say that process *p broadcasts message m* if p invokes $\mathit{rbroadcast}(m)$. We assume that every broadcast message m includes the following fields: the identity of its sender, denoted $\mathit{sender}(m)$, and a sequence number, denoted $\mathit{seq}(m)$. These fields make every message unique. We say that *q delivers message m* if q returns from the invocation of $\mathit{rdeliver}(m)$. Primitives *rbroadcast* and *rdeliver* satisfy the following properties:[6]

Validity: If a correct process broadcasts a message m, then it eventually delivers m.

Uniform Agreement: If a process delivers a message m, then all correct processes eventually deliver m.

Uniform Integrity: For every message m, every process delivers m at most once, and only if m was previously broadcast by $\mathit{sender}(m)$.

[5] We assume this set allows any process to make any query at any time.

[6] All the broadcast primitives that we define in this paper are *uniform* [10]. To abbreviate the notation, we drop the word "uniform" from the various broadcast types.

Validity and Uniform Agreement imply that if a correct process broadcasts a message m, then all correct processes eventually deliver m.

2.2 Atomic broadcast

Intuitively, atomic broadcast ensures that processes agree on the order they deliver messages. More precisely, *atomic broadcast* is defined in terms of primitives *abroadcast*(m) and *adeliver*(m) that must satisfy the Validity, Uniform Agreement and Uniform Integrity properties above, and the following property:

Uniform Total Order: If some process delivers message m before message m', then a process delivers m' only after it has delivered m.[7]

2.3 Generic broadcast

Generic broadcast is parametrized by a *conflict* relation (denoted $\sim$) defined over the set of messages; this relation is assumed to be symmetric and non-reflexive. Informally, generic broadcast ensures that if two messages m and m' conflict, then they are delivered in the same order. Messages that do not conflict are not required to be ordered. More precisely, *generic broadcast* is defined in terms of the conflict relation (given as a parameter) and two primitives: *gbroadcast*(m) and *gdeliver*(m) that must satisfy the Validity, Uniform Agreement and Uniform Integrity properties above, and the following property:

Uniform Generalized Order: If messages m and m' conflict and some process delivers m before m', then a process delivers m' only after it has delivered m.

If the conflict relation includes all the pairs of messages, generic broadcast coincides with atomic broadcast; if the conflict relation is empty, generic broadcast reduces to reliable broadcast.

3 Thrifty implementations

Let $\mathcal{A}$ be an implementation of generic broadcast that can use an oracle X, and let $\mathit{Runs}(\mathcal{A})$ be the set of runs of A. Let $\mathit{gbcast\_msgs}(r)$ be the set of messages gbroadcast in r and $\mathit{gbcast\_msgs}(r, [t, \infty))$ be the set of messages gbroadcast in r at or after time t.

Definition 1. *We say that* $\mathcal{A}$ *is* non-trivial w.r.t. oracle X *if, when no conflicting messages are* gbroadcast*,* X *is not used. More precisely:* $\forall r \in \mathit{Runs}(\mathcal{A})$, $[\forall m, m' \in \mathit{gbcast\_msgs}(r), m \not\sim m'] \Rightarrow X$ *is not used in* r.

Definition 2. *We say that* $\mathcal{A}$ *is* thrifty w.r.t. X *if it is non-trivial w.r.t.* X *and it guarantees the following property: if there is a time after which messages gbroadcast do not conflict with each other, then eventually* X *is no longer used. More precisely:* $\forall r \in \mathit{Runs}(\mathcal{A}), [\exists t, \forall m, m' \in \mathit{gbcast\_msgs}(r, [t, \infty)), m \not\sim m'] \Rightarrow \exists t', X$ *is not used in* r *after time* t'.

[7] In [10], Uniform Total Order is a weaker property.

4 A non-trivial implementation of generic broadcast

We now give a non-trivial implementation of generic broadcast for asynchronous systems with a majority of correct processes. The implementation, given in Figure 1, uses atomic broadcast as an oracle, and reliable broadcast as a subroutine (reliable broadcast can be easily implemented in asynchronous systems with process crashes without the use of oracles). In this implementation, $C(m)$ denotes the set $\{m\} \cup \{m' : m' \text{ conflicts with } m\}$.

To gbroadcast a message m, the basic idea is that processes go through two rounds of messages, and then the broadcaster p decides to either rbroadcast m (in which case the oracle is not used) or abroadcast m (in which case the oracle is used). More precisely, to gbroadcast a message m, p sends $(m, \textsc{first})$ to all processes, where FIRST is a tag to distinguish different types of messages. When a process receives $(m, \textsc{first})$, it adds m to its set *seen* of messages, and checks if m conflicts with any messages in *seen*. If it does, it sends $(m, \textsc{bad}, \textsc{second})$ to all processes; else, it sends $(m, \textsc{good}, \textsc{second})$. When a process receives a message of form $(m, *, \textsc{second})$ from $n - f$ processes, it adds m to its *seen* set, and then checks if a majority of SECOND messages are GOOD, and if its *seen* set has no messages conflicting with m. If so, the process adds m to its set *possibleRB*, and then sends $(m, \mathit{possibleRB} \cap C(m), \textsc{third})$ to p — the process that gbroadcast m — where $\mathit{possibleRB} \cap C(m)$ is the subset of messages in *possibleRB* that either conflict with m or is equal to m (note that $\mathit{possibleRB} \cap C(m)$ can be empty, it can contain m, and it can contain messages distinct from m). When p receives messages of the form $(m, \mathit{poss}, \textsc{third})$ from $n-f$ processes, it checks if a majority of them has m it its *poss* set. If so, p rbroadcasts m; else, p abroadcasts m, together with the union of all *poss* sets received. When a process rdelivers m, it gdelivers m if it has not done so previously. When a process adelivers (m, prec), it gdelivers all messages in *prec* (if it has not done so already), and then gdelivers m.

In this implementation, each process keeps two local variables: *seen* and *possibleRB*. The first one keeps the set of gbroadcast messages that the process has seen so far, and the second keeps the set of gbroadcast messages that are possibly reliably broadcast.

Theorem 1. *Consider an asynchronous system with a majority of correct processes ($n > 2f$). The algorithm in Figure 1 is a non-trivial implementation of generic broadcast that uses atomic broadcast as an oracle.*

Observation: In asynchronous systems with $n \leq 2f$, there are no non-trivial implementations of generic broadcast w.r.t. any oracle X.

5 A thrifty implementation of generic broadcast

We now give a thrifty implementation of generic broadcast that uses atomic broadcast as an oracle. It works in asynchronous systems in which a majority of processes is correct. This implementation is given in Figure 2, and builds

```
For every process $p$:

    initialization:
        $seen \leftarrow \emptyset$; $possibleRB \leftarrow \emptyset$

    to $gbroadcast(m)$: send $(m, \textsc{first})$ to all processes

    upon receive$(m, \textsc{first})$ from $q$:
        $seen \leftarrow seen \cup \{m\}$
        if $seen$ has no messages conflicting with $m$
        then send $(m, \textsc{good}, \textsc{second})$ to all processes
        else send $(m, \textsc{bad}, \textsc{second})$ to all processes

    upon receive$(m, *, \textsc{second})$ from $n - f$ processes for the first time:
        $seen \leftarrow seen \cup \{m\}$
        $good \leftarrow \{r :$ received $(m, \textsc{good}, \textsc{second})$ from $r\}$
        if $|good| > n/2$ and $seen$ has no messages conflicting with $m$
        then $possibleRB \leftarrow possibleRB \cup \{m\}$
        send $(m, possibleRB \cap C(m), \textsc{third})$ to $sender(m)$

    upon receive$(m, *, \textsc{third})$ from $n - f$ processes for the first time:
        $R \leftarrow \{r :$ received $(m, *, \textsc{third})$ from $r\}$
        for each $r \in R$ do $poss[r] \leftarrow M$ s.t. received $(m, M, \textsc{third})$ from $r$
        if $|r : m \in poss[r]| > n/2$ then $rbroadcast(m)$
        else $abroadcast(m, \cup_{r \in R} poss[r])$

    upon $rdeliver(m)$: if $m$ not $gdelivered$ then $gdeliver(m)$

    upon $adeliver(m, prec)$:
        for each $m' \in prec$ do
            if $m'$ not $gdelivered$ then $gdeliver(m')$
        if $m$ not $gdelivered$ then $gdeliver(m)$
```

Fig. 1. Non-trivial implementation of generic broadcast with an atomic broadcast oracle

upon the non-trivial implementation given in Section 4. In this implementation, $C(M) = M \cup \{m' : m'$ conflicts with some $m \in M\}$, and $C(m) = \{m\} \cup \{m' : m'$ conflicts with $m\}$.

Each process p keeps four variables: *seen*, *possibleRB*, *stable* and *adel*. *seen* is the set of gbroadcast messages that p has seen but has not yet adelivered or rdelivered. *possibleRB* is the set of gbroadcast messages that can be rbroadcast, but were not yet adelivered or rdelivered. *adel* is the set of messages that p has adelivered. *stable* is a set of pairs (m, B), where m is a message and B is a set of messages. Intuitively, $(m, B) \in stable$ means that p has adelivered or rdelivered m, and p must gdeliver all messages in B before it gdelivers m. We denote by π_1 the projection on the first component of a tuple or of a set of tuples. That is, $\pi_1((m, B))$ is m and $\pi_1(stable)$ is the set of m such that $(m, B) \in stable$, for some B.

To gbroadcast a message m, a process p sends $(m, \textsc{first})$ to all processes. Upon receipt of such a message, a process q adds m to its *seen* set, if m is not in $\pi_1(stable)$. Then q sends to all processes a SECOND message containing m, together with *seen*, and those elements of *stable* whose first component either conflicts with some message in $seen \cup \{m\}$ or belongs to $seen \cup \{m\}$. When a process q receives $(m, s, st, \textsc{second})$, it adds to *seen* those elements in s that are not in $\pi_1(stable)$, and it adds st to *stable*. When q collects SECOND messages from $n - f$ processes, it checks if *seen* contains m and no messages conflicting with m, and if so, q adds m to *possibleRB*. Then, q sends to p — the gbroadcaster of m — a THIRD message containing m, *seen*, *possibleRB* and those elements in *stable* whose first component either conflicts with some message in $seen \cup \{m\}$ or belongs to $seen \cup \{m\}$. When p receives a THIRD message for m from $n - f$ processes, it checks if a majority of them have m in their third components and if m is not in $\pi_1(stable)$. If so, p rbroadcasts m, together with those messages in $\pi_1(stable)$ that conflict with m. Else, p abroadcasts m, together with (1) the so-called *flush* set, which contains those messages that are in the *seen* sets of a majority of processes, (2) the so-called *prec* set, which contains those messages that are in the *possibleRB* set of some process and that either conflict with a message in $flush \cup \{m\}$ or belong to $flush \cup \{m\}$, (3) those messages in $\pi_1(stable)$ that either conflict with a message in $flush \cup prec \cup \{m\}$ or belong to $flush \cup prec \cup \{m\}$. We assume that, before p abroadcasts $(m, flush, prec, \ldots)$, p chooses some arbitrary ordering for the messages in *flush* and *prec*, which will be known to any process that adelivers $(m, flush, prec, \ldots)$.

When a process q rdelivers $(m, before)$, it removes m from *possibleRB* and from *seen*, and adds $(m, before)$ to *stable*. Then, q looks for elements (m', B) in *stable* such that m' has not been gdelivered and all messages in B have been gdelivered. If q finds such an element, q gdelivers m'.

When q adelivers $(m, flush, prec, before)$, it removes $\{m\} \cup prec \cup flush$ from *possibleRB* and from *seen*, and adds *adel* to *before*. Then, q iterates over the ordered elements of *prec*. For each element m' of *prec*, q adds to *stable* the tuple (m', B), where B is the elements in *before* that conflict with m'. The intuition here is that $(m', B) \in stable$ means that p must gdeliver m' *after* p gdelivers all elements in B. Then, in a similar fashion, q iterates over the ordered elements of *flush*, to add each of them to *stable*. Next, q adds m to *stable* and adds $\{m\} \cup flush \cup prec$ to *adel*. Finally, q looks for elements (m', B) in *stable* such that m' has not been gdelivered and all messages in B have been gdelivered. If q finds such an element, q gdelivers m'.

Theorem 2. *Consider an asynchronous system with a majority of correct processes ($n > 2f$). The algorithm in Figure 2 is a thrifty implementation of generic broadcast that uses atomic broadcast as an oracle.*

6 A sparing implementation of atomic broadcast

As we explained in the introduction, we would like to solve atomic broadcast with an algorithm that does not rely on an oracle whenever possible. Since no oracle is needed to order the delivery of causally related messages, we would like the atomic broadcast algorithm to stop using the oracle when messages are causally related.

More precisely, we say that *message m immediately causally precedes message m' and denote $m \rightarrow^1 m'$* if either (1) some process p abroadcasts m and then abroadcasts m' or (2) some process p adelivers m and then abroadcasts m'. Thus, $\rightarrow^1$ is a relation on the set of messages. Let $\rightarrow$ be the transitive closure

```
For every process $p$:

    initialization:
        $seen \leftarrow \emptyset$; $possibleRB \leftarrow \emptyset$; $stable \leftarrow \emptyset$; $adel \leftarrow \emptyset$

    to $gbroadcast(m)$: send $(m, \textsc{first})$ **to** all processes

    **upon** receive$(m, \textsc{first})$ **from** $q$:
        **if** $m \notin \pi_1(stable)$ **then** $seen \leftarrow seen \cup \{m\}$
        send $(m, seen, \{X \in stable : \pi_1(X) \in C(seen \cup \{m\})\}, \textsc{second})$ **to** all processes

    **upon** receive$(m, s, st, \textsc{second})$ **from** $q$:
        $seen \leftarrow seen \cup (s \setminus \pi_1(stable))$; $stable \leftarrow stable \cup st$
        **if** received messages of the form $(m, *, *, \textsc{second})$ from $n - f$ processes for the first time **then**
            **if** $m \notin \pi_1(stable)$ **then** $seen \leftarrow seen \cup \{m\}$
            **if** $seen \cap C(m) = \{m\}$ **then** $possibleRB \leftarrow possibleRB \cup \{m\}$
            send $(m, seen, possibleRB, \{X \in stable : \pi_1(X) \in C(seen \cup \{m\})\}, \textsc{third})$ **to** $sender(m)$

    **upon** receive$(m, *, *, *, \textsc{third})$ **from** $n - f$ processes for the first time:
        $R \leftarrow \{r :$ received $(m, *, *, *, \textsc{third})$ from $r\}$
        **for each** $r \in R$ **do**
            $s[r] \leftarrow M$, where $M$ is the set such that $p$ received $(m, M, *, *, \textsc{third})$ from $r$
            $poss[r] \leftarrow M$, where $M$ is the set such that $p$ received $(m, *, M, *, \textsc{third})$ from $r$
            $stable \leftarrow stable \cup M$, where $M$ is the set such that $p$ received $(m, *, *, M, \textsc{third})$ from $r$
        **if** $|r : m \in poss[r]| > n/2$ and $m \notin \pi_1(stable)$ **then** $rbroadcast(m, \pi_1(stable) \cap C(m))$
        **else if** $m \notin \pi_1(stable)$ **then**
            $flush \leftarrow \{m' : m' \neq m \wedge |q : m' \in s[q]| > n/2\}$
            $prec \leftarrow \cup_{r \in R} poss[r] \cap C(flush \cup \{m\})$
            $abroadcast(m, flush, prec, \pi_1(stable) \cap C(flush \cup prec \cup \{m\}))$
            /* in the abroadcast message above, sets $flush$ and $prec$ are ordered, arbitrarily */

    **upon** $rdeliver(m, before)$:
        $possibleRB \leftarrow possibleRB \setminus \{m\}$; $seen \leftarrow seen \setminus \{m\}$
        $stable \leftarrow stable \cup \{(m, before)\}$
        **while** $\exists (m', B) \in stable$ s.t. $m'$ not $gdelivered$ and all messages in $B$ have been $gdelivered$
        **do** $gdeliver(m')$

    **upon** $adeliver(m, flush, prec, before)$:
        $possibleRB \leftarrow possibleRB \setminus (\{m\} \cup prec \cup flush)$; $seen \leftarrow seen \setminus (\{m\} \cup prec \cup flush)$
        $before \leftarrow before \cup adel$
        **for each** $m' \in prec$ **do** $stable \leftarrow stable \cup \{(m', C(m') \cap before)\}$; $before \leftarrow before \cup \{m'\}$
        **for each** $m' \in flush$ **do** $stable \leftarrow stable \cup \{(m', C(m') \cap before)\}$; $before \leftarrow before \cup \{m'\}$
        /* the **for each** loops above iterate in the order of the ordered sets $prec$ and $flush$ */
        $stable \leftarrow stable \cup \{(m, C(m) \cap before)\}$; $adel \leftarrow adel \cup \{m\} \cup flush \cup prec$
        **while** $\exists (m', B) \in stable$ s.t. $m'$ not $gdelivered$ and all messages in $B$ have been $gdelivered$
        **do** $gdeliver(m')$
```

Fig. 2. Thrifty implementation of generic broadcast with an atomic broadcast oracle

of $\rightarrow^1$. We say that m *is causally related to* m' if either $m \rightarrow m'$ or $m' \rightarrow m$. If m and m' are not causally related, we say that m *and* m' *are concurrent.* These definitions are based on [11].

Let $\mathcal{A}^X$ be an implementation of atomic broadcast that uses an oracle X.

Definition 3. *We say that* $\mathcal{A}^X$ *is* sparing w.r.t. oracle X *if it guarantees the following property: if there is a time after which messages abroadcast are pairwise causally related, then eventually* X *is no longer used. More precisely:* $\forall r \in \mathit{Runs}(\mathcal{A}^X), [\exists t, \forall m, m' \in \mathit{gbcast\_msgs}(r, [t, \infty)), m \rightarrow m' \vee m' \rightarrow m] \Rightarrow \exists t', X$ *is not used in* r *after time* t'.

In this section, we show how to transform *any* implementation of atomic broadcast that uses some oracle X, into an implementation that is sparing w.r.t to X. As a first step, we show how to transform any implementation of generic broadcast that is thrifty w.r.t. oracle X, into an implementation of atomic broadcast that is sparing w.r.t. X. This is achieved through the algorithm in Figure 3.

In this algorithm, *seq* denotes the number of messages that p has abroadcast so far, while $\mathit{ndel}[q]$ is the number of messages from q that p has adelivered so far. Intuitively, *ts* is a vector timestamp for messages such that if *ts* is the timestamp of m, then $\mathit{ts}[j]$ is the number of messages from process j that causally precede m. We can show that if m causally precedes m' and their timestamps are *ts* and ts', respectively, then $\mathit{ts} \leq \mathit{ts}'$.

To abroadcast a message m, process p first obtains a new vector timestamp *ts* for m, by copying the vector *ndel* to *ts*, and then changing $\mathit{ts}[p]$ to a new sequence number. Then p gbroadcasts m with its timestamp *ts*. Upon gdeliver of (m, ts), a process q copies *ts* to *prec*, and changes $\mathit{prec}[\mathit{sender}(m)]$ to $\mathit{ts}[\mathit{sender}(m)] - 1$. Intuitively, *prec* represents the number of messages from each process that q must adeliver before q can adeliver m. Then q appends (m, prec) to L, and then searches for the first message (m', prec') in L with $\mathit{prec}' \leq \mathit{ndel}$.[8] If it finds such a message, it adelivers m', increments $\mathit{ndel}[\mathit{sender}(m')]$ by one, and removes (m', prec') from L.

Theorem 3. *Consider an asynchronous system with at least one correct process. If we plug-in an implementation of generic broadcast that is thrifty w.r.t. oracle* X *into the algorithm in Figure 3, then we obtain an implementation of atomic broadcast that is sparing w.r.t. oracle* X.

As we now explain, we can use this result to transform any implementation $\mathcal{A}^X$ of atomic broadcast that uses an oracle X, into an implementation $\mathcal{A}^X_{\mathit{sparing}}$ that is sparing w.r.t. X. To do so, we first replace the atomic broadcast oracle in Figure 2 with $\mathcal{A}^X$, and thus obtain an implementation $\mathcal{G}^X_{\mathit{thrifty}}$ of generic broadcast that is thrifty w.r.t. X. We then use the transformation in Figure 3 to transform $\mathcal{G}^X_{\mathit{thrifty}}$ to $\mathcal{A}^X_{\mathit{sparing}}$ — an implementation of atomic broadcast that is sparing w.r.t. X (by Theorem 3).

Theorem 4. *Given any implementation of atomic broadcast that uses some oracle* X*, we can transform it to one that is sparing w.r.t.* X.

[8] We say that a vector $v_1 \leq v_2$ if for every $q \in \Pi$, $v_1[q] \leq v_2[q]$.

```
For every process $p$:

  initialization:
    $seq \leftarrow 0$        /* # of messages abroadcast by $p$ */
    $L \leftarrow \emptyset$        /* ordered set with message to adeliver */
    for each $q \in \Pi$ do $ndel[q] \leftarrow 0$
                    /* $ndel[q]$ = # of messages from $q$ that $p$ has adelivered */
    define $(m, ts) \sim (m', ts')$ iff $ts \not\leq ts'$ and $ts' \not\leq ts$
                    /* conflict relation for generic broadcast */

  to $abroadcast(m)$:
    $seq \leftarrow seq + 1$; $ts \leftarrow ndel$; $ts[p] \leftarrow seq$        /* get new timestamp */
    $gbroadcast(m, ts)$        /* with $\sim$ as the conflict relation */

  upon $gdeliver(m, ts)$:
    $prec \leftarrow ts; prec[sender(m)] \leftarrow ts[sender(m)] - 1$
    $L \leftarrow L \cdot (m, prec)$        /* append $(m, prec)$ to $L$ */
    while $\exists (m', prec') \in L$ such that $prec' \leq ndel$ do
      $(m', prec') \leftarrow$ first element in $L$ such that $prec' \leq ndel$
      $adeliver(m')$
      $ndel[sender(m')] \leftarrow ndel[sender(m')] + 1$
      $L \leftarrow L \setminus (m', prec')$
```

Fig. 3. Transforming thrifty generic broadcast into sparing atomic broadcast

7 Low-latency thrifty implementations of generic broadcast

It is easy to see that the generic broadcast implementations in Figures 1 and 2 guarantee that in "good" runs with no failures and no conflicting messages, every message is delivered within 4δ, where δ is the maximum network message delay.[9] It turns out that we can decrease this latency to 3δ with some simple modifications to the algorithms. Moreover, if we assume that $n > 3f$ (i.e., more than two-thirds of the processes are correct) then we can further reduce the latency to 2δ. With the thrifty implementation, this latency is eventually achieved even in runs with failures and conflicting messages, provided that there is a time after which the messages gbroadcast are not conflicting.

Reducing the message latency to 3δ. To achieve a latency of 3δ in good runs, we modify the implementation in Figure 1 as follows: (1) processes should send the THIRD message to all processes in line 15, (2) instead of rbroadcasting a message m in line 19, a process p sends a message telling all processes to "deliver m", and then p gdelivers m, and (3) upon the receipt of a "deliver m" message for the first time, a process relays this "deliver m" message to all processes and gdelivers m. With this modification, it is easy to see that in good runs, every gbroadcast message is gdelivered within 3δ.

Theorem 5. *With the modifications above, the algorithm in Figure 1 ensures that, in runs with no failures and no conflicting messages, every gbroadcast message is gdelivered within 3δ, where δ is the maximum network message delay.*

[9] This assumes a reasonable implementation of reliable broadcast, which is used as a subroutine in these implementations.

We can modify the thrifty implementation in Figure 2 in a similar manner: (1) processes send the THIRD message to all processes in line 13, (2) instead of rbroadcasting a message in line 20, a process sends a "deliver $(m, \pi_1(\mathit{stable}) \cap C(m))$" message to all processes, sets variable *before* to $\pi_1(\mathit{stable}) \cap C(m)$, and then executes the code in lines 26–29, and (3) upon the receipt of message "deliver (m, before)" for the first time, a process relays this "deliver" message to all processes, and executes the code in lines 26–29.

Theorem 6. *With the modifications above, the algorithm in Figure 2 ensures that if there is a time after which the messages gbroadcast are not conflicting, eventually every gbroadcast message is gdelivered within* 3δ*, where* δ *is the maximum network message delay.*

Reducing the message latency to 2δ when $n > 3f$. To achieve a latency of 2δ in good runs, we assume that $n > 3f$ (instead of $n > 2f$). With this assumption, Figure 4 gives a non-trivial implementation of generic broadcast. The implementation is a simplification of the one in Figure 1, and uses atomic broadcast as the oracle.

```
For every process $p$:

    initialization:
        $seen \leftarrow \emptyset$; $good \leftarrow \emptyset$

    to $gbroadcast(m)$: send $(m, \text{FIRST})$ to all processes

    upon receive$(m, \text{FIRST})$ from $q$:
        $seen \leftarrow seen \cup \{m\}$
        if $seen \cap C(m) = \{m\}$ then $good \leftarrow good \cup \{m\}$
        send $(m, good \cap C(m), \text{SECOND})$ to all processes

    upon receive$(m, *, \text{SECOND})$ from $n - f$ processes for the first time:
        $R \leftarrow \{r : \text{received } (m, *, \text{SECOND}) \text{ from } r\}$
        for each $r \in R$ do $g[r] \leftarrow M$ s.t. received $(m, M, \text{SECOND})$ from $r$
        if $|\{r : g[r] = \{m\}\}| > 2n/3$ then send $(m, \text{DELIVER})$ to all processes; $gdeliver(m)$
        else if $p = sender(m)$ then
            $poss \leftarrow \{m' \neq m : |\{r : m' \in g[r]\}| > n/3\}$
            $abroadcast(m, poss)$

    upon receive$(m, \text{DELIVER})$ from some process:
        if $m$ not $gdelivered$ then send $(m, \text{DELIVER})$ to all processes; $gdeliver(m)$

    upon $adeliver(m, prec)$:
        for each $m' \in prec$ do
            if $m'$ not $gdelivered$ then $gdeliver(m')$
        if $m$ not $gdelivered$ then $gdeliver(m)$
```

Fig. 4. Low-latency non-trivial implementation of generic broadcast

Theorem 7. *Consider an asynchronous system with* $n > 3f$*. The algorithm in Figure 4 is a non-trivial implementation of generic broadcast that uses atomic broadcast as an oracle. In runs with no failures and no conflicting messages, every gbroadcast message is gdelivered within* 2δ*, where* δ *is the maximum network message delay.*

Figure 5 gives a thrifty implementation of generic broadcast with a latency of 2δ in good runs. The implementation is a simplification of the one in Figure 2, and uses atomic broadcast as the oracle.

Note that, in line 18, process p sends a message to itself. We did this to avoid repetition of code; p should not really send a message to itself, but rather execute the code in lines 24–28.

```
1  For every process $p$:

2     initialization:
3        $seen \leftarrow \emptyset$; $good \leftarrow \emptyset$; $stable \leftarrow \emptyset$; $adel \leftarrow \emptyset$

4     to $gbroadcast(m)$: send $(m, \text{FIRST})$ to all processes

5     upon receive$(m, \text{FIRST})$ from $q$:
6        if $m \notin \pi_1(stable)$ then $seen \leftarrow seen \cup \{m\}$
7        if $m \notin \pi_1(stable)$ and $seen \cap C(m) = \{m\}$ then $good \leftarrow good \cup \{m\}$
8        send $(m, seen, good, \{X \in stable : \pi_1(X) \in C(seen \cup \{m\})\}, \text{SECOND})$ to all processes

9     upon receive$(m, ss, *, st, \text{SECOND})$ from $q$:
10       $seen \leftarrow seen \cup (ss \setminus \pi_1(stable))$; $stable \leftarrow stable \cup st$
11       if received messages of the form $(m, *, *, *, \text{SECOND})$ from $n - f$ processes for the first time then
12          $R \leftarrow \{r :$ received $(m, *, *, *, \text{SECOND})$ from $r\}$
13          for each $r \in R$ do
14             $s[r] \leftarrow M$, where $M$ is the set such that $p$ received $(m, M, *, *, \text{SECOND})$ from $r$
15             $g[r] \leftarrow M$, where $M$ is the set such that $p$ received $(m, *, M, *, \text{SECOND})$ from $r$
16          if $m \notin \pi_1(stable)$ then $seen \leftarrow seen \cup \{m\}$
17          if $|r : m \in g[r]| > 2n/3$ and $m \notin \pi_1(stable)$
18          then send $(m, \pi_1(stable) \cap C(m), \text{DELIVER})$ to $p$
19          else if $m \notin \pi_1(stable)$ and $p = sender(m)$ then
20             $flush \leftarrow \{m' : m' \neq m \wedge |q : m' \in s[q]| > 2n/3\}$
21             $prec \leftarrow \{m' : m' \neq m \wedge |q : m' \in g[q]| > n/3\}$
22             $abroadcast(m, flush, prec, \pi_1(stable) \cap C(flush \cup prec \cup \{m\}))$
               /* in the abroadcast message above, sets $flush$ and $prec$ are ordered, arbitrarily */

23    upon receive$(m, before, \text{DELIVER})$ from some process for the first time:
24       send $(m, before, \text{DELIVER})$ to all processes
25       $good \leftarrow good \setminus \{m\}$; $seen \leftarrow seen \setminus \{m\}$
26       $stable \leftarrow stable \cup \{(m, before)\}$
27       while $\exists (m', B) \in stable$ s.t. $m'$ not $gdelivered$ and all messages in $B$ have been $gdelivered$
28       do $gdeliver(m')$

29    upon $adeliver(m, flush, prec, before)$:
30       $good \leftarrow good \setminus (\{m\} \cup prec \cup flush)$; $seen \leftarrow seen \setminus (\{m\} \cup prec \cup flush)$
31       $before \leftarrow before \cup adel$
32       for each $m' \in prec$ do $stable \leftarrow stable \cup \{(m', C(m') \cap before)\}$; $before \leftarrow before \cup \{m'\}$
33       for each $m' \in flush$ do $stable \leftarrow stable \cup \{(m', C(m') \cap before)\}$; $before \leftarrow before \cup \{m'\}$
         /* the for each loops above iterate in the order of the ordered sets $prec$ and $flush$ */
34       $stable \leftarrow stable \cup \{(m, C(m) \cap before)\}$; $adel \leftarrow adel \cup \{m\} \cup flush \cup prec$
35       while $\exists (m', B) \in stable$ s.t. $m'$ not $gdelivered$ and all messages in $B$ have been $gdelivered$
36       do $gdeliver(m')$
```

Fig. 5. Low-latency thrifty implementation of generic broadcast with an atomic broadcast oracle

Theorem 8. *Consider an asynchronous system with $n > 3f$. The algorithm in Figure 5 is a thrifty implementation of generic broadcast that uses atomic broadcast as an oracle. If there is a time after which the messages gbroadcast are not conflicting, eventually every gbroadcast message is gdelivered within 2δ, where δ is the maximum network message delay.*

References

1. M. K. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg. Thrifty generic broadcast. Technical Report (to appear), DIX, École Polytechnique, Palaiseau, France, 2000.
2. G. Alvarez, F. Cristian, and S. Mishra. On-demand fault-tolerant atomic broadcast protocol. In *Proceedings of the Fifth IFIP International Conference on Dependable Computing for Critical Applications*, Sept. 1995.
3. Y. Amir, P. Moser, L.E. Melliar-Smith, D. Agarwal, and P. Ciarfella. The totem single-ring ordering and membership protocol. *ACM Trans. Comput. Syst.*, 13(4):311–342, Nov. 95.
4. K. P. Birman and T. A. Joseph. Reliable communication in the presence of failures. *ACM Trans. Comput. Syst.*, 5(1):47–76, Feb. 1987.
5. T. D. Chandra and S. Toueg. Unreliable failure detectors for asynchronous systems. *J. ACM*, 43(2):225–267, Mar. 1996.
6. W. Chen, S. Toueg, and M. K. Aguilera. On the quality of service of failure detectors. In *Proceedings of the First International Conference on Dependable Systems and Networks (also FTCS-30)*, June 2000.
7. F. Cristian, H. Aghili, H. R. Strong, and D. Dolev. Atomic broadcast: From simple message diffusion to Byzantine agreement. In *Proceedings of the Fifteenth International Symposium on Fault-Tolerant Computing*, pages 200–206, June 1985.
8. D. Dolev, S. Kramer, and D. Malki. Early delivery totally ordered multicast in asynchronous environment. In *IEEE Proceedings of the 23th International Symp on Fault-tolerant computing (FTCS-23)*, pages 544–553, June 1993.
9. A. Gopal, R. Strong, S. Toueg, and F. Cristian. Early-delivery atomic broadcast (extended abstract). In *Proceedings of the Ninth ACM Symposium on Principles of Distributed Computing*, pages 297–309, Aug. 1990.
10. V. Hadzilacos and S. Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical Report TR 94-1425, Cornell University, Dept. of Computer Science, Cornell University, Ithaca, NY 14853, May 1994.
11. L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.
12. F. Pedone and A. Schiper. Generic broadcast. In *Proceedings of the 13th International Symposium on Distributed Computing (DISC'99, formerly WDAG)*, Sept. 1999.
13. F. Pedone and A. Schiper. Generic broadcast. Technical Report SSC/1999/012, École Polytechnique Fédérale de Lausanne, Switzerland, Apr. 1999.
14. R. van Renesse, Y. Minsky, and M. Hayden. A gossip-style failure detection service. In *Proceedings of Middleware'98*, Sept. 1998.