Technical Report

UCAM-CL-TR-638
ISSN 1476-2986

Number 638

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Optimistic Generic Broadcast

Piotr Zieliński

July 2005

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

Technical reports published by the University of Cambridge
Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/TechReports/

ISSN 1476-2986

Optimistic Generic Broadcast

Piotr Zieliński
University of Cambridge
Computer Laboratory
piotr.zielinski@cl.cam.ac.uk

Abstract

We consider an asynchronous system with the Ω failure detector, and investigate the number of communication steps required by various broadcast protocols in runs in which the leader does not change. Atomic Broadcast, used for example in state machine replication, requires three communication steps. Optimistic Atomic Broadcast requires only two steps if all correct processes receive messages in the same order. Generic Broadcast requires two steps if no messages conflict. We present an algorithm that subsumes both of these approaches and guarantees two-step delivery if all conflicting messages are received in the same order, and three-step delivery otherwise. Internally, our protocol uses two new algorithms. First, a Consensus algorithm which decides in one communication step if all proposals are the same, and needs two steps otherwise. Second, a method that allows us to run infinitely many instances of a distributed algorithm, provided that only finitely many of them are different. We assume that fewer than a third of all processes are faulty ($n > 3f$).

1 Introduction

State machine replication [7] is a common way of increasing fault-tolerance of client-server systems. As opposed to centralized systems, where clients send requests to a single server, in this approach, each request is broadcast to a group of servers. In order for this approach to work, client requests must be delivered to all the servers in the same order. The broadcast primitive that ensures this property is called Uniform Atomic Broadcast[1].

We consider an asynchronous system with the Ω leader oracle [3], and investigate the number of communication steps required by various broadcast protocols in runs in which the leader does not change. Atomic Broadcast is expensive in the sense that it requires three communication steps: one for the client request to reach the servers and two for the servers to agree on the order of the requests [4, 10, 12]. Two approaches have been proposed in the literature to deal with this problem: Optimistic Atomic Broadcast and Generic Broadcast.

[1] "Uniform" abstractions offer guarantees to all processes as opposed to only correct ones. All abstractions considered in this paper are uniform, so from now on we will omit this word from abstraction names.

Algorithm	no conflicting messages	all messages same order	conflicting messages same order	general scenario
Chandra and Toueg [3]	3	3	3	3
Opt. Atomic Broadcast [14]	4	2	4	4
Generic Broadcast [16]	2	4	4	4
This paper	2	2	2	3

Figure 1: Latency degree comparison of several broadcast algorithms.

Optimistic Atomic Broadcast [14] takes advantage of the fact that in many networks messages tend to arrive at different processes in the same order. Protocols implementing this primitive are able to deliver messages in two communication steps as long as all processes receive messages in the same order.

Generic Broadcast [1, 16] introduces a binary conflict relation on messages, and requires only that conflicting messages are delivered in the same order by all processes. For example, two "read" requests or any two requests operating on unrelated objects are non-conflicting and can be delivered in different orders. In runs without conflicting messages, Generic Broadcast implementations can deliver all messages in two steps.

In this paper, we present an algorithm that implements both of these approaches at the same time. Optimistic Generic Broadcast, as we call it, guarantees message delivery in two communication steps if all correct processes receive all *conflicting* messages in the same order. When this condition does not hold, all messages are delivered within three communication steps. Both of these latencies match the respective lower bounds [10, 16]. Our algorithm requires that fewer than a third of all processes are incorrect ($n > 3f$), which is necessary to deliver messages in two steps [15].

Fig. 1 gives a short comparison of several algorithms. We consider four scenarios: without conflicting messages, with correct processes receiving *all* messages in the same order, with correct processes receiving *conflicting* messages in the same order, and with no restrictions. For each of these scenarios, Fig. 1 shows the number of communication steps required to deliver messages, formally known as *latency degree* [17]. The comparison shows that our algorithm is, in terms of latency, strictly better than the other protocols. This remains true even if we ignore the third column, which is specific to our algorithm.

Our Generic Broadcast protocol uses a new Consensus algorithm. If the leader is correct and does not change, this algorithm decides in one communication step if all proposals are the same, and in two steps otherwise. In the literature, algorithms have been proposed that satisfy only the first condition [2], only the second condition [3, 9, 11, 17], or both but the first only for the privileged value [6]. To the best of our knowledge, the Consensus algorithm presented in this paper is the first to meet both conditions fully.

The paper is structured in the following way. Section 2 introduces our asynchronous system model, and gives precise definitions of Consensus, Atomic Broadcast, and Generic Broadcast. Sections 3 and 4 describe our Optimistic Generic Broadcast algorithm. Section 5 presents the implementation of One-Two Consensus. Finally, Section 6 shows how to implement infinitely many Consensus instances at the same time, which is used by our Generic Broadcast algorithm. Correctness of the presented algorithms is only argued informally; for rigorous proofs, see the appendix.

2 System Model

Our system model consists of n processes, out of which at most f can fail by crashing. We assume that less than a third of all the processes are faulty ($n > 3f$). Processes communicate through asynchronous reliable channels, that is, there is no time limit on message transmission time, and messages between correct processes never get lost.

We assume that each process is equipped with an unreliable leader oracle Ω, which eventually outputs the same correct leader at all correct processes. Theorem A.1 shows that Ω is the weakest failure detector that allows us solve (Optimistic) Generic Broadcast in an asynchronous system, provided at least one pair of messages conflict. In the special case when no messages conflict, Generic Broadcast becomes identical to Uniform Reliable Broadcast [8].

In the Consensus problem, processes submit proposals and are required to eventually agree on one of them. Formally, we require the following properties:

Uniform Validity. If a process decides on x, then some process proposed x.

Uniform Agreement. No two processes decide differently.

Termination. If all correct processes propose, then they will eventually decide.

In Atomic Broadcast, processes broadcast messages, which are delivered by all processes in the same order. Formally [5],

Validity. If a correct process broadcasts a message m, then all correct processes will eventually deliver m.

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

Uniform Integrity. For any message m, every process delivers m at most once, and only if m was previously broadcast.

Uniform Total Order. If some process delivers message m' after message m, then every process delivers m' only after it has delivered m.

Generic Broadcast is identical to Atomic Broadcast, except that only conflicting messages must be delivered in the same order:

Uniform Partial Order. If some process delivers message m' after message m *conflicting with* m', then every process delivers m' only after it has delivered m.

The notion of conflict is captured by a binary conflict relation on the set of all possible messages, which is a parameter of the problem [16]. For example, one might consider a relation on *read* and *write* requests in which all pairs of messages conflict unless both of them are *read*s. We assume that the conflict relation and therefore the (infinite) set of all possible messages are both known to all processes in advance [1, 16].

3 Basic Generic Broadcast Algorithm

We say that a run is *stable* if the (correct) leader output by the Ω failure detector is the same at all correct processes and never changes. In this section, we will present a simplified version of our Generic Broadcast algorithm, which is correct in stable runs with correct senders but might not make progress in other runs. In Section 4, we will show how to extend this algorithm to obtain a fully correct Generic Broadcast implementation.

3.1 Partial Order on Messages

Our algorithm operates by making processes agree on the delivery order of each pair of conflicting messages. More precisely, processes cooperate in building a partial order "$\rightarrow$" on conflicting messages and deliver messages in any order consistent with this partial order. For any two messages m and m', relation $m \rightarrow m'$ requires m to be delivered before m'. Since non-conflicting messages can be delivered in any order, the relation "$\rightarrow$" is defined only for pairs of messages $\{m, m'\}$ that conflict. For these, we expect that eventually either $m \rightarrow m'$ or $m' \rightarrow m$.

The following diagram shows an example with four messages. All pairs of messages conflict, except for m_2 and m_3, which can be delivered in different orders by different processes.

Processes deliver messages in any order consistent with the partial order "$\rightarrow$". In the first example, "$\rightarrow$" is defined for all pairs of conflicting messages. Processes can deliver the four messages in one of two orders m_1, m_2, m_3, m_4 or m_1, m_3, m_2, m_4, both consistent with "$\rightarrow$". Messages m_2 and m_3 can be delivered in different orders by different processes; this does not violate the Partial Order property because these messages do not conflict.

In the second example, the relation between conflicting messages m_2 and m_4 is not known (yet). As a result, none of them can be delivered. However, whatever the order of m_2 and m_4 will be, one of the orders: m_1, m_3, m_2, m_4 and m_1, m_3, m_4, m_2 will be consistent with "$\rightarrow$". These two orders share a common prefix m_1, m_3, so messages m_1 and m_3 can be delivered straight away.

Another way of looking at the delivery process is realizing that m_1 can be delivered because $m_1 \rightarrow m$ for all $m \neq m_1$. After m_1 has been delivered, we can deliver m_3 because $m_3 \rightarrow m$ for all undelivered m's conflicting with m_3. Conflicting messages m_2 and m_4 will be delivered only when their order is known.

3.2 Basic Algorithm

In our algorithm, processes agree on the partial order "$\rightarrow$" by using Consensus to agree on the delivery order of every pair of conflicting messages. In other words, for each *unordered* pair $\{m, m'\}$ of conflicting messages, we use a separate instance of Consensus. In each

```
1  when receive(m) do
2    for all possible non-received messages m' conflicting with m do
3      first_{m,m'}.propose(m)
4  when first_{m,m'}.decide(m) do
5    set m → m'
6  when m is undelivered and
7        m → m' for all undelivered messages m' conflicting with m do
8    deliver(m)
```

Figure 2: Basic Generic Broadcast algorithm.

such instance $first_{m,m'}$, each process p proposes the message m or m' that arrived at p first.

The resulting partial order is built based on decisions of the Consensus instances $first_{m,m'}$. If the instance decides on m, then m is deemed to be the first message of the two ($m \rightarrow m'$). Hence, if the instance $first_{m,m'} = first_{m',m}$ decides on m', then $m' \rightarrow m$. Messages are delivered in an order consistent with "$\rightarrow$". Section 4 will explain why cycles do not appear in stable runs, and explain how to deal with them in other runs.

The basic algorithm is shown in Fig. 2. Senders broadcast their messages using ordinary broadcast. When a process receives a message m, it proposes m to instances $first_{m,m'}$ for all messages m' conflicting with m that have not been received (yet). In other words, the process proposes m to precede all such messages m' in the delivery order. An undelivered message m is delivered once $m \rightarrow m'$ holds for all possible undelivered messages m' conflicting with m.

Since the set of all messages is usually infinite, the $receive(m)$ action involves executing infinitely many instances of Consensus. Section 6 will show how to accomplish this with finite resources. Until then, we will stick with the "infinite" version of the algorithm because it is easier to understand.

The algorithm satisfies Uniform Integrity and Uniform Partial Order. For the former, we assume the existence of an artificial message $\perp$, which is never sent and conflicts with all other messages. Therefore, delivering any message m requires $m \rightarrow \perp$, which means that some process must have proposed m to $first_{m,\perp}$, so some process must have broadcast m.

To prove Uniform Partial Order, we will assume, to derive a contradiction, that conflicting messages m and m' are delivered in different orders at different processes. This would mean that $m \rightarrow m'$ at one of the processes, and $m' \rightarrow m$ at another, which is impossible by the Uniform Agreement property of the underlying Consensus.

3.3 Delivery Latency

In this section, we will explain why, in stable runs, the basic algorithm delivers all messages in three communication steps. We will also see that if, in addition, all conflicting messages arrive at all correct processes in the same order, then all these messages are delivered in two communication steps.

We assume that the underlying Consensus algorithm satisfies the following two prop-

erties:

C1: In stable runs in which all correct processes propose the same value, the decision is made one communication step after all correct processes proposed.

C2: In stable runs, all correct processes decide on the value proposed by the leader two communication steps after the leader proposed (even if other processes have not proposed).

Section 5 presents a Consensus algorithm with these properties.

Any message broadcast by a correct process is received by the leader in one communication step. Property **C2** ensures that the order is decided two communication steps later, giving three communication steps in total for delivery latency. If all correct processes receive conflicting messages in the same order, then they propose the same order to Consensus instances. Therefore, Property **C1** implies that decision will be made in one communication step, giving two communication steps in total for delivery latency.

4 Full Generic Broadcast Algorithm

Before presenting the full version of the algorithm, we will highlight two main problems with the basic version in unstable runs.

4.1 Cycles

Cycles in the relation "$\rightarrow$" built by the basic algorithm lead to a deadlock. In stable runs, cycles do not appear because Property **C2** ensures that "$\rightarrow$" reflects the linear order of message reception at the leader. In unstable runs, however, different parts of the "$\rightarrow$" relation might have been proposed by different processes.

As an example, consider a system with three processes p_1, p_2, p_3. Each of these processes receives three messages m_1, m_2, m_3 in a different order:

$$\begin{aligned} p_1 &: m_1, m_2, m_3 && \text{executes } \mathit{first}_{m_1,m_2}.\mathit{propose}(m_1), \\ p_2 &: m_2, m_3, m_1 && \text{executes } \mathit{first}_{m_2,m_3}.\mathit{propose}(m_2), \\ p_3 &: m_3, m_1, m_2 && \text{executes } \mathit{first}_{m_3,m_1}.\mathit{propose}(m_3). \end{aligned}$$

If all of the above proposals become decisions, then the cycle

$$m_1 \rightarrow m_2 \rightarrow m_3 \rightarrow m_1$$

will be formed, and as a result none of these messages will ever be delivered.

To cope with cycles, we introduce the notion of blocked messages. We say that a message is *blocked* if it belongs to a cycle, or it is a successor of a blocked message. In our example, all three messages are blocked, and any message m_4 with, say, $m_2 \rightarrow m_4$ would be blocked as well. Obviously, blocked messages will never be delivered by the basic algorithm. In the full version of the algorithm, we will sometimes deliver blocked messages to break cycles and avoid deadlocks.

4.2 Faulty Senders

Another problem with the basic algorithm are faulty senders. If a sender crashes, then its messages might reach only a subset of the correct processes. For Consensus instances, this means that not all the correct processes propose, and therefore it can happen that only some of the correct processes decide. Since these decisions are directly related to message deliveries, the Uniform Agreement property of Generic Broadcast might be violated.

A common solution to this problem is to make processes rebroadcast every received message, thereby implementing Non-Uniform Reliable Broadcast [7]. Thus, if a correct process receives a message, then all correct processes eventually will. As a result, all correct processes will propose, decide, and deliver the message.

Therefore, to ensure Uniform Agreement, it is sufficient to guarantee that every delivered message has been received by at least one correct process. For this reason we introduce the notion of a process "seeing" a message. We say that a process *sees* a message m if it has received, in some instance of Consensus, a message containing m (either directly from the sender or from other processes). Recall that a message is delivered only if at least one Consensus instance decided on it. Since the decision of any Consensus instance must have been seen by at least one *correct* process (Lemma A.2), every delivered message has been seen by a correct process, which is exactly what we need.

The problem discussed here could also be solved by making the senders use *Uniform* Reliable Broadcast [8]. This abstraction, however, requires two communication steps, which would slow our algorithm down by one step.

4.3 Algorithm

Fig. 3 shows the full version of our algorithm. To resolve cycles and be able to deliver blocked messages, we use an auxiliary Atomic Broadcast protocol. The latency of this Atomic Broadcast protocol affects only unstable runs because cycles cannot appear in stable ones.

A process reacts only to messages m received for the first time. It rebroadcasts m to other processes using both ordinary broadcast and also Atomic Broadcast. The rationale behind the former is to ensure that if one correct process receives a message, then eventually all correct processes will. As mentioned before, Atomic Broadcast is used to resolve cycles. Finally, as in the basic version, the process executes $\mathit{first}_{m,m'}.\mathit{propose}(m)$ for all possible messages m' conflicting with m that were not received before m. The order "$\rightarrow$" is constructed in the same way as in the basic version.

In the full version, messages can be delivered either normally or during cycle resolution. To distinguish these two kinds of deliveries, we call the former 1-delivery, and the latter 2-delivery. Messages are 1-delivered in exactly the same way as in the basic version.

If a process has seen a message, it behaves as if it had received it. The decision of any Consensus instance must have been seen by at least one *correct* process. Therefore, any 1-delivered message has been seen by a correct process, who broadcast it, so that all correct processes would eventually receive that message and propose it to some Consensus instances. This property is vital for Uniform Agreement.

The cycle resolution task loops over messages delivered by the underlying Atomic Broadcast protocol. For each such message m, the task executes $\mathit{see}(m)$ and waits until one of the two conditions holds. If m has already been delivered, then the loop goes to

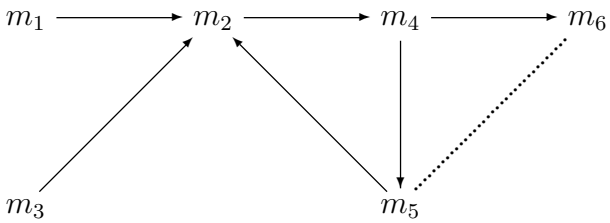
```
1   when receive(m) do
2     if m is received for the first time then
3       broadcast(m)
4       abcast(m)
5       for all possible non-received messages m' conflicting with m do
6         first_{m,m'}.propose(m)
7   when see(m) do
8     receive(m)
9   when first_{m,m'}.decide(m) do
10    set m → m'
11  when m is undelivered and
12    m → m' for all undelivered messages m' conflicting with m do
13    deliver_1(m)
14  task cycle-resolution is
15    repeat forever
16      wait until abdeliver(m)
17      see(m)
18      wait until m has been delivered or
19                  all undelivered messages conflicting with m are blocked
20      if m has not been delivered yet then
21        deliver_2(m)
```

Figure 3: Full Generic Broadcast algorithm.

the next iteration. Otherwise, if all undelivered messages conflicting with m are blocked, then m is delivered. The rationale behind this strategy is that, since none of the blocked messages can be 1-delivered, it is safe to deliver m, thereby, possibly, breaking the cycle. The use of Atomic Broadcast ensures that the messages m chosen to break cycles are the same at different processes.

To show that lines 18–19 always terminate, we must prove that all never-blocked messages m' conflicting with m will eventually be delivered. Since m' is not blocked, the graph of paths $m' \leftarrow m_1 \leftarrow \cdots \leftarrow m_k$ of undelivered messages does not contain cycles, and forms a tree rooted at m'. The leaves of this tree will be successively 1-delivered, resulting eventually in 1-delivery of m'.

4.4 Example



Both m_1 and m_3 can be 1-delivered because the only message conflicting with them (m_2) is their successor. Messages m_1 and m_3 can be delivered in any order; this does not violate Partial Order because they do not conflict. Assume that, at the same time, the *cycle-resolution* task abdelivers m_1. Since the only message conflicting with it (m_2) is blocked by the cycle $m_2 \rightarrow m_4 \rightarrow m_5 \rightarrow m_2$, message m_1 is 2-delivered if it has not been 1-delivered yet.

After m_1 and m_3 have been delivered, no other messages can be 1-delivered, because all of them are blocked. If now the cycle resolution task abdelivers m_2, then since all undelivered messages conflicting with it (m_4, m_5) are blocked, m_2 is 2-delivered. This delivery breaks the cycle, and now all undelivered messages conflicting with m_4 (i.e., m_5 and m_6) are its successors, so m_4 is 1-delivered. Conflicting messages m_5 and m_6 cannot be delivered until the processes agree on their order. If, for example, $m_5 \rightarrow m_6$, then message m_5 will be 1-delivered, followed by m_6.

5 One-Two Consensus

Our Generic Broadcast algorithm uses One-Two Consensus shown in Fig. 4. This algorithm uses three underlying Consensus instances: 1, 2, and L. When a process p proposes x, it first broadcasts the pair (x, p), and then uses instance 1 to propose x, instance 2 to propose the pair (x, p), and instance L to propose the current output of its leader oracle Ω. Instances 1 and L make a decision in one communication step, provided that all correct processes propose the same value. We achieve this by using the algorithm by Brasileiro et al. [2], which requires $n > 3f$. Instance 2 decides in two communication steps, however, it requires only the correct leader to have proposed. This can be implemented with the Paxos algorithm [11].

When process p wants to decide, it first waits until instance L decides on some leader l. In stable runs, in which the output of Ω is the same at all correct processes, this

```
1   when propose(x) by process p do
2     broadcast(x, p)
3     propose_1(x)
4     propose_2(x, p)
5     propose_L(l) where l is the leader output by Ω
6   task decide at process p is
7     wait until decide_L(l)
8     wait until at least one of the conditions is true and decide on x
9       condition 1: decide_1(x) and receive(x, l)
10      condition 2: decide_2(x, l)
11      condition 3: decide_1(x) and decide_2(y, q) with q ≠ l
```

Figure 4: One-Two Consensus.

should happen within one communication step. Then, process p waits until at least one of the three conditions in Fig. 4 holds. The first two conditions correspond to instances 1 and 2 deciding on the value proposed by the leader. If all correct processes propose the same value, then the first condition will hold in one communication step. If the leader is correct, then the second condition will hold in two communication steps. The third condition is a fall-back designed for unstable runs; processes adopt the decision from instance 1, provided that instance 2 did not decide on the value proposed by the leader.

5.1 Thriftiness

Aguilera et al. [1] define a Generic Broadcast algorithm to be *thrifty* if it uses the underlying oracle (here Ω) only when some conflicting messages are received. Our algorithm is not thrifty; the one-two step Consensus described above uses Ω in all cases. To make our algorithm thrifty, we need a Consensus algorithm that, if all correct processes propose the same value, decides in one step, *without* using Ω. This property is a stronger version of **C1**, which we call **C1***.

The algorithm by Brasileiro et al. [2] satisfies **C1***; it decides in one step if all correct processes propose the same value, without using Ω, otherwise it starts an underlying Consensus algorithm. This means that it does not satisfy **C2**; if correct processes propose different values, it may take three steps to decide, instead of two. For our Optimistic Generic Broadcast algorithm this means four steps in general stable runs instead of three. In terms of latency, such an algorithm is still strictly better than both Generic Broadcast [1, 13, 16] and Optimistic Atomic Broadcast [14], however might be slower than the one proposed by Chandra and Toueg [3] (Fig. 1).

Guerraoui and Raynal [6] proved that no Consensus algorithm based on Ω can be both configuration-efficient and oracle-efficient, which in our case implies that Properties **C1*** and **C2** cannot hold at the same time. This observation makes us conjecture that no thrifty Generic Broadcast algorithm can achieve the latencies from the bottom row in Fig. 1.

The underlying Atomic Broadcast protocol, employed by our algorithm to break cycles, also uses Ω [3]. However, in the absence of conflicting messages, cycles cannot be

```
1  I is the family of disjoint sets of virtual instances, initially empty
2  when receive event e tagged with the set I_e do
3    for each I ∈ I do
4      split I into I ∩ I_e and I \ I_e
5    create a new physical instance A_I for I = I_e \ ⋃I, and add I to I
6    eliminate empty sets from I
7    for each I ∈ I do
8      if I ∩ I_e ≠ ∅ then
9        send the event e to instance A_I
```

Figure 5: Emulating infinitely many virtual instances.

created. Therefore, instead of processes performing $abcast(m)$ immediately after receiving m (as in Fig. 3), they can defer it until they have received some conflicting messages. This modification prevents the algorithm from using Atomic Broadcast if no conflicting messages are received.

6 Handling Infinitely Many Instances

Our Generic Broadcast algorithm uses infinitely many Consensus instances. This section explains how to implement an infinite number of *virtual instances* of a distributed algorithm using only finitely many *physical instances* at every process. As we will see, this is possible provided that there are only finitely many *different* virtual instances. For example, in our Generic Broadcast algorithm from Fig. 3, a process proposes *the same* message m to an infinite number of instances of Consensus.

Let us start with finitely many virtual instances, and denote these by i_1, i_2, ..., i_k. The usual approach is to tag any event (a message or a function call) with the identifier of the instance. Each process runs k physical instances of the algorithm. Every event tagged with i_k is directed to the k-th instance, and every event produced by the k-th instance is tagged with i_k. In this case, virtual and physical instances are the same, so this method can be used only with finitely many virtual instances.

To implement an infinite number of virtual instances, some of them must share a single physical instance. Algorithm in Fig. 5 maintains a family $\mathcal{I}$ of disjoint sets of virtual instances, initially empty. Each element $I \in \mathcal{I}$ is a set of virtual instances that share a single physical instance denoted as A_I. All events generated by A_I are tagged with the set I.

When an event tagged with a set of virtual instances I_e arrives, the process does the following. First, if some virtual instances sharing the same physical instance start to differ, the physical instance is cloned. This is done by splitting elements $I \in \mathcal{I}$ into $I \cap I_e$ and $I \setminus I_e$, so that every element of $\mathcal{I}$ is either a subset of I_e or disjoint with it. When such a split happens, the physical instance A_I is replaced with two new instances $A_{I \cap I_e}$ and $A_{I \setminus I_e}$, both identical to A_I. Also, a new physical instance is created for $I_e \setminus \bigcup \mathcal{I}$ to ensure that every virtual instance corresponds to some physical instance; in other words, we want to make sure that for each $i \in I_e$ there is $I \in \mathcal{I}$ with $i \in I$. Finally, the event is sent to all physical instances corresponding to any virtual instances in I_e.

6.1 Representing Sets

The above method can be used to execute an infinite number of Consensus instances at the same time, provided that we can represent infinite sets of instances in a finite form. For use in the algorithm from Fig. 5, the families of representable sets must be closed under subtraction and intersection. Closeness under set union is not necessary; $I_e \setminus \bigcup \mathcal{I}$ can be computed by iteratively subtracting elements of $\mathcal{I}$ from I_e. In this section, we will briefly present such representations for some families of sets useful in our Generic Broadcast algorithm.

Border sets.

Finite sets can be trivially represented by listing their elements. The family of *border sets* contains all sets that are either finite (F) or are complements of finite sets ($\overline{F}$). For example, the set $\overline{\{m_1, m_2\}}$ consists of all messages except for m_1 and m_2. The representation of a border set consists of the finite set F and a flag indicating whether the set is F or $\overline{F}$. The family of border sets is closed under negation and intersection (which implies subtraction):

$$F_1 \cap F_2 = F_1 \cap F_2, \qquad F_1 \cap \overline{F_2} = F_1 \setminus F_2,$$
$$\overline{F_1} \cap F_2 = F_2 \setminus F_1, \qquad \overline{F_1} \cap \overline{F_2} = \overline{F_1 \cup F_2}.$$

M-sets.

If only messages m_1 and m_2 have been received, then the border set $\overline{\{m_1, m_2\}}$ represents the set of all non-received messages. Can we use border sets to represent more complex sets such as "the set of all non-received messages conflicting with m"? The answer depends on the conflict relation. It is often the case that messages can be divided into a small number of categories (e.g., "read" and "write"), such that conflict properties of messages are determined by the categories they belong to. Consider a system with k categories C_1, ..., C_k, where C_i is the set of all messages in the i-th category. For any border sets B_1, ..., B_k satisfying $B_i \subseteq C_i$, we define a *m-set*

$$\langle B_1, \ldots, B_k \rangle = B_1 \cup \cdots \cup B_k$$

to be the set containing all messages from sets B_1, B_2, ..., B_k.

As an example consider a system with two categories: "read" and "write"; any two requests conflict unless they are both reads. Assume that requests w_1, w_2, r_1, and r_2 have been received. The set of all non-received requests conflicting with r_2 is $\langle \emptyset, \overline{\{w_1, w_2\}} \rangle$, that is, no read requests and all possible write requests except for w_1 and w_2.

The family of m-sets is closed under subtraction and intersection:

$$\langle B_1, \ldots, B_k \rangle \cap \langle B'_1, \ldots, B'_k \rangle = \langle B_1 \cap B'_1, \ldots, B_k \cap B'_k \rangle,$$
$$\langle B_1, \ldots, B_k \rangle \setminus \langle B'_1, \ldots, B'_k \rangle = \langle B_1 \setminus B'_1, \ldots, B_k \setminus B'_k \rangle.$$

Sets of message pairs.

In our Generic Broadcast algorithm, each Consensus instance is identified by an unordered pair of messages. By $\{m, M\}$ we denote the set of pairs containing m and one element of

the m-set M:

$$\{m, M\} = \{\, \{m, m'\} : m' \in M \,\}$$

For example, M can be the set of all possible non-received messages m' conflicting with a given message m. Consider the *receive*(m) routine from our Generic Broadcast algorithm in Fig. 3. We can replace infinitely many invocations of $\mathit{first}_{m,m'}.\mathit{propose}(m)$ with a single $\mathit{first}_{m,M}.\mathit{propose}(m)$. The family of sets $\{m, M\}$ can be used in the algorithm from Fig. 5 because it is closed under intersection and subtraction (we assume $m \neq m'$):

$$\begin{aligned}
\{m, M\} \cap \{m, M'\} &= \{m, M \cap M'\}, \\
\{m, M\} \setminus \{m, M'\} &= \{m, M \setminus M'\}, \\
\{m, M\} \cap \{m', M'\} &= \begin{cases} \{m, \{m'\}\} & \text{if } m \in M' \text{ and } m' \in M, \\ \{m, \emptyset\} & \text{otherwise,} \end{cases} \\
\{m, M\} \setminus \{m', M'\} &= \begin{cases} \{m, M \setminus \{m'\}\} & \text{if } m \in M' \text{ and } m' \in M, \\ \{m, M\} & \text{otherwise.} \end{cases}
\end{aligned}$$

7 Conclusion

We presented a new algorithm that solves the Generic Broadcast problem, in which conflicting messages must be delivered in the same order by all processes. In stable runs, our algorithm delivers messages in three communication steps. If all conflicting messages are received by all correct processes in the same order, then all messages are delivered within two communication steps. In terms of latency degree [17], this algorithm is strictly better than any proposed so far [1, 3, 13, 14, 16], and matches several lower bounds [10, 15, 16]. Although we implicitly assumed that only the main n processes are allowed to broadcast messages, our model can easily be extended to allow external processes to broadcast as well.

In our algorithm, for each pair of conflicting messages, processes use Consensus to agree on their order. Messages are delivered in any order consistent with the agreed partial order. In unstable runs, circular dependencies can occur, and these are broken using Atomic Broadcast. The abstract formulation of the algorithm uses infinitely many instances of Consensus at the same time. Since there are only finitely many *different* instances, this can be emulated with finite resources, as we show in this paper. The new underlying Consensus algorithm uses one-step [2] and two-step [11] algorithms to decide in one communication step if all proposals are the same, and in two otherwise (in stable runs).

Although our Generic Broadcast algorithm is optimal in terms of latency degree, its message and computation complexities may still be prohibitive. Therefore, we see our contribution in the lower-bound matching algorithm. Further research will be required to optimize it for practical applications.

References

[1] Marcos Kawazoe Aguilera, Carole Delporte-Gallet, Hugues Fauconnier, and Sam Toueg. Thrifty Generic Broadcast. *Lecture Notes in Computer Science*, 1914:268–

282, 2000.

[2] Francisco Brasileiro, Fabíola Greve, Achour Mostefaoui, and Michel Raynal. Consensus in one communication step. *Lecture Notes in Computer Science*, 2127:42–50, 2001.

[3] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.

[4] Bernadette Charron-Bost and André Schiper. Uniform Consensus is harder than Consensus. Technical Report DSC/2000/028, Swiss Federal Institute of Technology (EPFL), Lausanne, Switzerland, May 2000.

[5] Xavier Défago, André Schiper, and Péter Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Surveys*, 36(4):372–421, 2004.

[6] Rachid Guerraoui and Michel Raynal. The information structure of indulgent Consensus. Technical Report PI-1531, IRISA, April, 2003.

[7] Vassos Hadzilacos and Sam Toueg. Fault-tolerant broadcast and related problems. In Sape Mullender, editor, *Distributed Systems*, chapter 5, pages 97–146. ACM Press, New York, 2nd edition, 1993.

[8] Vassos Hadzilacos and Sam Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical Report TR94-1425, 1994.

[9] Michel Hurfin and Michel Raynal. A simple and fast asynchronous consensus protocol based on a weak failure detector. *Distributed Computing*, 12(4):209–223, 1999.

[10] Idit Keidar and Sergio Rajsbaum. On the cost of fault-tolerant Consensus when there are no faults. *ACM SIGACT News*, 32, 2001.

[11] Leslie Lamport. Paxos made simple. *ACM SIGACT News*, 32(4):18–25, December 2001.

[12] Leslie Lamport. Lower bounds on Consensus. Unpublished note, March 2002.

[13] Fernando Pedone and André Schiper. Handling message semantics with Generic Broadcast protocols. *Distributed Computing*, 15(2):97–107, 2002.

[14] Fernando Pedone and André Schiper. Optimistic Atomic Broadcast: a pragmatic viewpoint. *Theoretical Computer Science*, 291(1):79–101, 2003.

[15] Fernando Pedone and André Schiper. On the inherent cost of Generic Broadcast. Technical Report IC/2004/46, Swiss Federal Institute of Technology (EPFL), May 2004.

[16] Fernando Pedone and André Schiper. Generic Broadcast. In *Proceedings of the 13th International Symposium on Distributed Computing (DISC'99)*, 1999.

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

A Proofs

We will first make some definitions. Each process p builds its own relation "$\rightarrow$", which is also denoted as "$\rightarrow_p$" if p is not obvious from the context. This relation changes over time, which might lead to confusion in proofs. To avoid it, we assume that, unless explicitly said otherwise, the symbol "$\rightarrow$" represents the ultimate form of the relation, that is, the union of the relations $\rightarrow$ taken at all moments in time. If $m \rightarrow m'$, then we say that m is a *predecessor* of m', and m' is a *successor* of m. A finite *path* $m \quad m'$ is a sequence of messages $m = m_1 \rightarrow m_2 \rightarrow \cdots \rightarrow m_k = m'$. Recall that a message m has been *seen* if at least one *correct* process has received a message containing m in some Consensus instance.

Theorem A.1. *Failure detector Ω is the weakest failure detector that allows us to solve Generic Broadcast, provided that at least one pair of messages conflict.*

Proof. Let us denote these two conflicting messages by 0 and 1. Then, one can solve binary Consensus by having each participant broadcast its proposal using a Generic Broadcast algorithm, and adopting the first delivered message as the decision. Since Ω is necessary to solve (binary) Consensus in an asynchronous system [3], the same holds for Generic Broadcast. □

Lemma A.2. *Consider any Uniform Consensus algorithm. If a process decides on x, then at least one correct process has seen x.*

Proof. To obtain a contradiction, assume that there is a run r_1, in which a (possibly incorrect) process decides on x at time t, and no correct process ever sees x. Consider a run r_2, which is identical to r_1 except that all incorrect processes that have not failed by time t in run r_1 fail at time t. Also assume that, in run r_2, all messages from incorrect processes that have not reached their destination before time t are lost.

Runs r_1 and r_2 are identical until time t, so in r_2 no correct process sees x by time t. After time t, correct processes receive only messages from correct processes. Therefore, no correct process ever sees x in run r_2.

Assume that, in run r_2, every correct process that did not propose by time t, proposes at time t some $x' \neq x$. Since all correct processes propose in run r_2, the Termination property implies that all correct processes will eventually decide. They cannot decide on x, because they have never seen this value. On the other hand, Uniform Agreement implies that they must decide on x because some process has already done so. This contradiction proves the assertion. □

Lemma A.3. *If $m \rightarrow m'$, then m' has been seen by a correct process.*

Proof. Relation $m \rightarrow m'$ requires that Consensus instance $\mathit{first}_{m,m'}$ decided on m, so Lemma A.2 implies the assertion. □

Lemma A.4. *Any message m seen by a correct process has a finite number of predecessors.*

Proof. The Validity property of the underlying Consensus algorithm implies that for any predecessor m', at least one process must have executed $\mathit{first}_{m,m'}.\mathit{propose}(m')$. Therefore, at least one process received m' before m. We have to prove that there are only finitely many such messages m'.

Message m has been seen by a correct process, so all correct processes eventually receive m. Therefore, any correct process receives only finitely many messages m' before m. Incorrect processes receive finitely many messages before they crash. Therefore, the total number of messages m' that precede m is finite. □

Lemma A.5. *Let m be a message seen by a correct process. Eventually, $m \rightarrow m'$ or $m' \rightarrow m$ for any m' conflicting with m.*

Proof. The assumption implies that all correct processes will eventually receive m. Therefore, all correct processes will eventually propose in all instances $\mathit{first}_{m,m'}$ (this will happen when the process receives its first message in $\{m, m'\}$). Eventually, all such instances will decide, which implies the assertion. □

Lemma A.6. *A never-delivered message m seen by a correct process has a never-delivered predecessor.*

Proof. To obtain a contradiction, assume that every predecessor of m will eventually be delivered. Message m has a finite number of predecessors (Lemma A.4), so eventually all predecessors of m will be delivered. Lemma A.5 implies that eventually every message m' conflicting with m will be either its predecessor or successor. As a result, eventually all undelivered messages conflicting with m will be its successors, so m will be 1-delivered. This contradicts the assumption of m never being delivered. □

Lemma A.7. *All paths $m_1 \leftarrow m_2 \leftarrow m_3 \leftarrow \cdots$ are finite.*

Proof. Assume that the path $m_1 \leftarrow m_2 \leftarrow m_3 \leftarrow \cdots$ is infinite. Properties of the leader elector Ω ensure that eventually all processes will either crash or output a single process p as the leader. Let M be the (finite) set of messages received by any process before this happens. Consider an (infinite) tail $m_k \leftarrow m_{k+1} \leftarrow \cdots$ of the original path that does not contain any messages from M. Since all messages m_k, m_{k+1}, ... were received after the output of the leader elector stabilized, all relations $m_k \leftarrow m_{k+1} \leftarrow \cdots$ are consistent with the linear order of message reception at the eventual leader (Consensus Property **C2**) However, this means that the eventual leader received infinitely many messages before m_k, which is impossible. □

Lemma A.8. *If a correct process executes $\mathit{abdeliver}(m)$, then it will deliver m.*

Proof. To obtain a contradiction, assume that a correct process has abdelivered m but will never deliver it. Processes see each message they abdeliver, so Lemma A.6 implies that m has a never-delivered predecessor.

We will prove that m has a never-delivered predecessor m' that is never blocked. To obtain a contradiction, assume that all never-delivered predecessors m' will eventually be blocked. This implies (previous paragraph) that at least one of the predecessors of m is blocked, which implies that m itself is blocked. As a result, all successors of m are blocked

as well. Therefore, Lemma A.5 implies that eventually all undelivered messages conflicting with m will be blocked, so m will be 2-delivered. This contradicts the assumption of m never being delivered, and proves that m has a never-delivered, never-blocked predecessor m'.

Consider a set of all paths $m'' \quad m'$ consisting only of never-delivered messages. There is at least one such path ($m' \quad m'$). Lemma A.7 implies that all such paths are finite, so there is a maximal path $m'' \quad m'$ (otherwise we could keep extending any path ad infinitum).

Both messages m' and m'' have been seen by a correct process because they have a successor in the path $m'' \quad m' \rightarrow m$ (Lemma A.3). As a result, Lemma A.6 implies that m'' has a never-delivered predecessor m'''.

If $m''' \in m'' \quad m'$, then m''' is blocked because $m''' \rightarrow m'' \quad m'''$, and as a result m' is blocked as well, which contradicts the assumption that m' is never blocked. On the other hand, if $m''' \notin m'' \quad m'$, then the path $m''' \rightarrow m'' \quad m'$ contains only never-delivered messages and extends $m'' \quad m'$, which contradicts the maximality of $m'' \quad m'$. These contradictions prove the assertion. □

Theorem A.9 (Validity). *If a correct sender gbcasts a message m, then all correct processes will eventually deliver it.*

Proof. The assumption implies that all correct processes will eventually receive m and execute $abcast(m)$. Therefore, all correct processes will eventually execute $abdeliver(m)$, so Lemma A.8 implies the assertion. □

Theorem A.10 (Uniform Agreement). *If a process delivers m, then all correct processes will eventually deliver m.*

Proof. If m has been 1-delivered, Lemma A.2 implies that message m is has been seen by a correct process. This process will eventually execute $abcast(m)$, so all correct processes will eventually execute $abdeliver(m)$.

If m has been 2-delivered, then it has been abdelivered, therefore all correct processes will eventually execute $abdeliver(m)$. In both cases, Lemma A.8 implies the assertion. □

Theorem A.11 (Uniform Integrity). *A process delivers m only once and only if m was gbcast by some sender.*

Proof. No message can be delivered twice because delivery of a message requires it to have not been delivered before. If m is 1-delivered, then $first_{m,\perp}$ decides on m, which implies that some process proposed m, which implies the assertion. If m is 2-delivered, then it must have been abcast by some process, which also implies the assertion. □

A.1 Partial Order

Definition A.12. *A process "(1-,2-)delivers message m before m'" iff it (1-,2-)delivers m and at the time of this delivery m' has not been delivered. (Message m' can be delivered later in any way or not delivered at all.)*

Lemma A.13. *Assume that process p 2-delivers m before m' and process q 2-delivers m' before m. This is impossible.*

Proof. At the time of 2-delivery of m at p, message m' is not delivered. Therefore, process p abdelivers m before m'. By a similar argument, process q abdelivers m' before m. This violates the Uniform Total Order property of the underlying Atomic Broadcast protocol.□

Lemma A.14. *Let m and m' be conflicting messages. Assume that process p 1-delivers m before m' and process q 1-delivers m' before m. This is impossible.*

Proof. In order to 1-deliver m before m' at process p, we must have $m \rightarrow m'$. An analogous argument for process q, gives $m' \rightarrow m$, which violates Uniform Agreement of the underlying Consensus algorithm. □

Lemma A.15. *Let m and m' be conflicting messages. Assume that some process p 2-delivers m before m', and process q delivers m' before m. This is impossible.*

Proof. Consider the moment when process p executes $deliver_2(m)$. Let B be the set of undelivered messages blocked at p. This set contains all undelivered messages conflicting with m. We will prove that, at any process q, no message from B will be delivered before m.

To obtain a contradiction, assume that B is not empty, and q delivers the first message $m' \neq m$ from B before m. We shall now obtain a contradiction by proving that process q can neither 2-deliver nor 1-deliver m' before m. Note that m and m' do not necessarily conflict.

Process q 2-delivering m' before m violates Lemma A.13 because process p 2-delivers m before m'.

Message $m' \in B$ is blocked at p, therefore it has an undelivered, blocked predecessor m'' at p. In other words, there is $m'' \in B$ such that $m'' \rightarrow_p m'$. Note that m' is the first message in B delivered by q; thus, at the moment of 1-delivery of m', message $m'' \in B$ is still undelivered at q. This leads to a contradiction: 1-delivery of m' requires $m' \rightarrow_q m''$, which is impossible because $m'' \rightarrow_p m'$. □

Theorem A.16 (Partial Order). *If some process delivers message m' after message m* conflicting with *m', then every process delivers m' only after it has delivered m.*

Proof. To obtain a contradiction, assume that process p delivers m before m', and process q delivers m' before m. If process p 2-delivers m before m', then Lemma A.15 prevents process q from delivering m' before m. Therefore, process p 1-delivers m before m'. By the analogous argument, process q 1-delivers m' before m. However, this is impossible by Lemma A.14. □

A.2 Latency

In this section, we will assume stable runs and that "time" is defined is a way in which one communication step corresponds to d units of time (d is not known to the algorithm).

Lemma A.17. *In stable runs, if the leader receives a message at time t, then all correct processes will deliver it by time $t + 2d$.*

Proof. To obtain a contradiction, assume this is not true and let m be the first message received by the leader for which the assertion does not hold.

Let m' be any message conflicting with m that was not received by the leader before m (at time t). When the leader receives m, it executes $\mathit{first}_{m,m'}.\mathit{propose}(m)$ at time t. Therefore, by Property **C2** of underlying Consensus, by time $t + 2d$, all correct processes execute $\mathit{first}_{m,m'}.\mathit{decide}(m)$ and set $m \to m'$.

By assumption, all messages m' received by the leader before m were delivered before time $t + 2d$, therefore, at time $t + 2d$, the 1-delivery condition for m is met. □

Corollary A.18 (Three-step delivery). *Every message gbcast at time t by a correct process, will be delivered by time $t + 3d$.*

Lemma A.19. *Assume all correct processes receive all conflicting messages in the same order. Any message received by all correct processes by time t will be delivered by $t + d$.*

Proof. To obtain a contradiction, assume this is not true and let m be the first message received by the leader for which the assertion does not hold.

Let m' be any message conflicting with m that was not received by the leader before m. By assumption, no correct process receives m' before m. Therefore, all correct processes execute $\mathit{first}_{m,m'}.\mathit{propose}(m)$ by time t. As a result, Property **C1** of underlying Consensus implies that, by time $t + d$, all correct processes execute $\mathit{first}_{m,m'}.\mathit{decide}(m)$ and set $m \to m'$.

Let m' be any message conflicting with m received by the leader before m. By assumption, all correct processes receive m' before m, therefore before time t. By another assumption, m' will be delivered by time $t + d$. Therefore, at time $t + d$, the 1-delivery condition for m is met. □

Corollary A.20 (Two-step delivery). *If all correct processes receive all conflicting messages in the same order, then every message gbcast at time t by a correct process, will be delivered by time $t + 2d$.*

A.3 One-Two Consensus

Theorem A.21 (Uniform Agreement). *No two processes decide on different values.*

Proof. In most cases, this follows from the same property of auxiliary instances of Consensus. This property can be violated only if some process decides in condition 2, whereas another in condition 1 or 3. Decisions in conditions 1 and 2 must be the same because they are both values proposed by the leader l. Conditions 2 and 3 cannot be used in the same execution. Condition 2 is used only if instance 2 decides on a value proposed by the leader, whereas condition 3 is used only if instance 2 decides on a value proposed by another process. □

Theorem A.22 (Uniform Validity). *If a process decides on x, then x was proposed by some process.*

Proof. Follows from analogous properties of Consensus instances 1 and 2. □

Theorem A.23 (Termination). *If all correct processes propose, then eventually all correct processes will decide.*

Proof. Termination properties of the underlying instances of Consensus imply that eventually every correct process will execute $decide_L(l)$, $decide_1(x)$ and $decide_2(y, q)$. If $q = l$, then condition 2 will decide on y. Otherwise, condition 3 will decide on x. □

Theorem A.24 (Property C1). *In stable runs in which all correct processes propose the same value, the decision is made in one communication step after all correct processes proposed.*

Proof. The assumption ensures that all correct processes, including the leader, will propose the same value to instances 1 and L. As a result, the leader l is known and condition 1 holds in one communication step [2]. □

Theorem A.25 (Property C2). *In stable runs, all correct processes decide on the value proposed by the leader in two communication steps after the leader proposed.*

Proof. The assumption ensures that all correct processes will propose the same leader to instance L, so all correct processes will decide on the leader l in one communication step. The leader l is correct, so its proposal (x, l) will become the decision in two communication steps [11]. As a result, condition 2 will hold. □