Reliable and Total Order Broadcast in the Crash-Recovery Model*

Romain Boichat Rachid Guerraoui
Communication Systems Department
Swiss Federal Institute of Technology
CH - 1015 Lausanne

Abstract

This paper addresses the problem of broadcasting messages in a reliable and totally ordered manner when processes and channels may crash and recover, or crash and never recover. We present a suite of specifications of *reliable* and *total order* broadcast primitives and we describe algorithms that implement those specifications. Our approach is *modular* and *incremental*. It is *modular* in the sense that the properties of broadcast primitives are first given separately and then composed: this provides a comprehensive design space for broadcast semantics. It is *incremental* in the sense that a broadcast algorithm implementing a given specification is obtained by transforming an algorithm that implements a weaker specification: this gives an automatic way to improve the resilience of broadcast primitives. We derive specific reliable and total order broadcast algorithms and we discuss their performance and optimality.

Contact author: Romain Boichat.[1]

Keywords: reliable broadcast, total order broadcast, modularity, transformation, optimisation, crash-recovery model.

1 Introduction

Broadcasts primitives facilitate the development of distributed applications. We consider in this paper two of the most important of such primitives: *reliable broadcast* and *total order* broadcast. Both allow processes to broadcast messages with some reliability guarantees. Roughly speaking, reliable broadcast ensures that all processes agree on the *set* of messages they deliver, while total order broadcast ensures that all processes agree on the *sequence* of messages they deliver. In short, a total order broadcast is a reliable broadcast where processes deliver messages in the same order. This paper addresses the problem of devising algorithms[2] that implement reliable

*Some material in sections 3, 4.2 and 5 appeared in [6].

[1]DSC LPD, EPFL, CH-1015 Lausanne, Switzerland, e-mail: Romain.Boichat@epfl.ch, Phone/Fax: +41 21 693 6702/7570

[2]We focus here on deterministic algorithms, unlike [4] for instance which considers randomised algorithms that offer probabilistic guarantees.

and total order broadcast primitives assuming a practical asynchronous *crash-recovery* model: processes and channels may crash and recover or crash and never recover.

Motivation. Given their wide applicability, broadcast primitives have been extensively studied for over a decade. In particular, many papers have been published on algorithms that implement reliable and total order broadcast primitives in a *crash-stop* system model [10, 14, 3, 13, 5, 7]. According to this model, channels are reliable and processes execute the algorithm assigned to them, unless they crash, in which case they simply halt their activities. Processes that do not crash are called *correct* processes. The simplicity of this model was a key to studying and comparing many broadcast algorithms, and also devising rigorous proofs for their correctness.

The practicality of the crash-stop system model is however questionable. The assumption that some processes never crash, and that those that crash never recover, is indeed simple but is quite unrealistic. In practice, processes that crash eventually recover and resume their activities. In the meantime, i.e., between the crash and the recovery events, the messages sent to a crashed process are lost. After a crash, a process typically loses the content of its volatile memory and only preserves the content of its stable storage. Devising algorithms for the crash-recovery model is more tricky than for the *crash-stop* model, precisely because of the need of careful use of stable storage. Processes should log in stable storage crucial information that will help them recover in a consistent state, but performing a forced log[3] is expensive and should be avoided as much as possible.

In summary, there is a significant literature about *crash-stop resilient* broadcast algorithms, but these do not fit a more realistic crash-recovery model which introduces a non-trivial complexity through the use of stable storage. The motivation of our work is precisely to devise *crash-recovery resilient* broadcast primitives.

Specifications and implementations. The specification of a reliable broadcast primitive is composed of three kinds of properties [15]: a *validity* property (V) that ensures the liveness of the broadcast, an *agreement* property (A) which ensures consensus on message delivery, and an *integrity* property (I) that prevents the absence of spurious messages and multiple deliveries. The specification of a total order broadcast primitive contains an additional *total order* (TO) type of property [15].

Devising crash-recovery resilient broadcast primitives goes first through providing meaningful variants of those properties in a crash-recovery model. Indeed, the possibility for the processes

[3] A synchronous write on disk.

to crash and recover impacts the actual definition of the very notion of process correctness, and consequently requires to revisit the specifications of broadcast primitives, e.g., in comparison with the specifications initially defined for a crash-stop model [15]. As we show in this paper, several meaningful specifications are possible for every property of a crash-recovery resilient broadcast. In fact, in the context of a crash-recovery model, every property of a given kind (*validity*, *agreement*, *integrity* and *total order*) might come in different flavours, according to whether:

1. We only restrict the behaviour of the processes that do not crash: we end up with the weakest properties, denoted by *V.1, A.1, I.1*, and *TO.1*. For instance, agreement here (A.1) would not preclude the situation where a process p_i delivers a message before crashing and no other process ever delivers that message, even if p_i recovers and never crashes again.

2. We also restrict the behaviour of the processes that recover - and remain up for sufficiently long: we end up with stronger properties, denoted by *V.2, A.2, I.2*, and *TO.2*. Typically, agreement here (A.2) would prevent the situation above, but would not preclude the situation where a process p_i delivers a message before permanently crashing and no other process ever delivers that message.

3. We restrict the behaviour of all processes: we end up with the strongest properties, denoted by *V.3, A.3, I.3*, and *TO.3*. Agreement here (A.3) would ensure that if any process p_i delivers a message, every correct would deliver the message, even if p_i crashes just after delivering the message and never recovers.

This paper defines these properties in a precise manner and describes how they can be combined in various ways to obtain meaningful specifications of crash-recovery resilient broadcast primitives (reliable and total order broadcast). We first point out some interesting relationships between the specifications and we propose *transformer* algorithms that build upon a broadcast primitive that satisfies a given specification (e.g., *V.1, A.1, I.1*, and *TO.1*) to implement a broadcast primitive that satisfies a stronger specification (e.g., *V.2, A.2, I.2*, and *TO.2*). Our unit of broadcast transformation is the individual specification. Transformers for reliable broadcast, together with transformers for total order broadcast, are instances of the same generic algorithm. This genericity enables us to factor out some fundamental differences between reliable and total order broadcast in a crash-recovery model, while capturing their similarities. This promotes algorithm layering, e.g., along the lines of [16].

We give algorithms that implement our different specifications in an incremental manner. We start by considering crash-stop resilient broadcast algorithms, namely the reliable broadcast algorithm of [15] and the total order broadcast algorithm of [7]. We show how to slightly improve these algorithms to satisfy the weakest of our crash-recovery resilient specifications (V.1, A.1, I.1, TO.1). We then discuss the algorithms that result from applying our transformers to implement stronger specifications. We point out simple techniques to optimise these algorithms and we give corresponding lower bounds (in terms of forced logs). Practical performance measures are given to depict the actual differences between algorithms that implement different specifications.

Contributions. This paper aims at giving a comprehensive study of crash-recovery resilient broadcast specifications and possible implementations.

- We draw a sharp line between the specifications and the implementations of broadcast primitives. In particular, we define various forms of specifications for reliable broadcast and total order broadcast. To our knowledge, this is the first time such a suite of specifications is given in a crash-recovery model.

- We present a systematic way of strengthening the resilience of crash-recovery resilient broadcast primitives. We do so using generic transformer algorithms that do not make any assumptions on the underlying broadcast algorithms (as long as they implement their specifications).

- We give specific crash-recovery resilient broadcast algorithms that we obtain from transforming crash-stop resilient broadcast algorithms, namely the algorithms of [15] and [7]. Interestingly, our resulting algorithms have the same number of communication steps than the original crash-stop algorithms in *nice* runs, i.e., runs where processes are up and messages are not lost: these are the most frequent runs in practice. In other words, we point out the very fact that the price to pay for moving to a crash-recovery model is in terms of forced logs.

- We discuss simple techniques to optimise our algorithms in terms of forced logs, and we give some general lower bound results that match our algorithms. Our experimental study helps quantify the performance difference between algorithms implementing different specifications.

Roadmap. The rest of the paper is organised as follows. Section 2 describes our crash-recovery model. Section 3 defines the specifications of our crash-recovery resilient broadcast primitives.

Section 4 presents our transformer algorithms. Section 5 focuses on specific algorithms and discuss their performance from an analytical as well as an experimental point of view. Section 6 discusses related work and draws some concluding remarks. Due to a lack of space and given that they are close to those of reliable broadcast, the correctness proofs of our total order broadcast transformers and algorithms are given in optional Appendix A.

2 Model

2.1 Processes

We consider a set of processes $\Pi = \{p_1, p_2, ..., p_n\}$. At any given time, a process is either *up* or *down*. When it is *up*, a process progresses at its own speed behaving according to its specification (*i.e.*, it correctly executes its program). Note that we do not make here any assumption on the relative speed of processes. While being up, a process can fail by crashing; it then stops executing its program and becomes *down*. A process that is down can later recover; it then becomes up again and restarts by executing a recovery procedure. The occurrence of a *crash* (resp. *recovery*) event makes a process transit from up to down (resp. from down to up). We say that a process p_i is *unstable* if it crashes and recovers infinitely many times. We define an *always-up* process as a process that never crashes. We say that a process p_i is *correct* if there is a time after which the process is permanently up.[4] A process is *faulty* if it is not *correct*, i.e., either *eventually always-down* or *unstable*. We assume that once p_i recovers, p_i is reset to the state *initialised*.

A process is equipped with two local memories: a volatile memory and a stable storage. The primitives **store** and **retrieve** allow a process that is up to access its stable storage. When it crashes, a process loses the content of its volatile memory; the content of its stable storage is however not affected by the crash and can be retrieved by the process upon recovery. We assume the presence of a discrete global clock whose range ticks $\mathcal{T}$ is the set of natural numbers. This clock is used to simplify presentation and not to introduce time synchrony, since processes cannot access the global clock.

[4] In practice, a correct process is required to stay up long enough for the computation to terminate. In asynchronous systems however, characterising the notion of "long enough" is impossible.

2.2 Link Properties

Processes exchange information and synchronise by *sending* and *receiving* messages through fair-lossy channels. We assume the existence of a bidirectional channel between every pair of processes. We assume that every message m includes the following fields: the identity of its sender, denoted *sender(m)*, and a local identification number, denoted *id(m)*. These fields make every message unique. Channels can lose or drop messages and there is no upper bound on message transmission delays. We assume the same channel definition given in [1], which ensures the following properties between every pair of processes p_i and p_j:

No creation: If p_j receives a message m from p_i at time t, then p_i sent m to p_j before time t.

Finite duplication: If p_i sends a message m to p_j only a finite number of times, then p_j receives m only a finite number of times.

Fair loss: If p_i sends a message m to p_j an infinite number of times and p_j is correct, then p_j receives m from p_i an infinite number of times.

The last two properties are sometimes called, respectively, *finite duplication* and *weak loss*, e.g., in [19]. They reflect the usefulness of the communication channel. Without these properties, any interesting distributed problem would be trivially impossible to solve. By introducing the notion of correct process into the *fair loss* property, we define the conditions under which a message is delivered to its recipient process. Indeed, the delivery of a message requires the recipient process to be running at the time the channel attempts to deliver it, and therefore depends on the failure pattern occurring in the execution. The *fair loss* property indicates that a message can be lost, either because the channel may not attempt to deliver the message or because the recipient process may be down when the channel attempts to deliver the message to it. In both cases, the channel is said to commit an *omission failure*.

2.3 Retransmission Module

We introduce here a retransmission module that encapsulates retransmissions issues to deal with temporary crashes of communication channels. This module is a basic block underlying our algorithms (see Section 5). The primitives of the retransmission module (*s-send* and *s-receive*) preserve the no creation and finite duplication properties of the underlying channels, and ensures the following validity property:

Validity: Let p_i be any process that s-sends a message m to a process p_j, and then p_i does not crash. If p_j is correct, then p_j eventually s-receives m.

Figure 1 gives the algorithm of the retransmission module. All messages that need to be retransmitted are put in the variable *xmitmsg* with their destination in the set *dst* (line 5). Messages in *xmitmsg* are erased once all recipients have acknowledged m, otherwise they are always retransmitted (lines 18-21).

```
 1: for each process p_i:
 2: procedure initialisation:
 3:    xmitmsg[], dst[] ← ⊥; start task{retransmit}
 4: procedure s-send(m)                                   {to s-send m to p_j}
 5:    if m ∉ xmitmsg then xmitmsg ← xmitmsg ∪ m
 6:    if p_j ∉ dst[m] then dst[m] ← dst[m] ∪ p_j
 7:    for all p_j ∈ dst[m] do
 8:       if p_j ≠ p_i then
 9:          send m to p_j
10:       else
11:          simulate receive m from p_i
12: upon receive(m) from p_j do
13:    if m = ACK then
14:       dst[m] ← dst[m] \p_j
15:       if dst[m] = ⊥ then xmitmsg ← xmitmsg \m
16:    else
17:       s-receive(m); send ACK(m) to p_j
18: task retransmit                                      {retransmit all messages}
19:    while true do
20:       for all m ∈ xmitmsg do
21:          s-send(m)
```

Figure 1: Retransmission module

Proposition 1. *Validity: Let p_i be any process that s-sends a message m to a process p_j, and then p_i does not crash. If p_j is correct, then p_j eventually s-receives m.*

Proof. Suppose that p_i s-sends a message m to a process p_j and then p_i does not crash. Assume by contradiction that p_j is correct, yet p_j does not s-receive m. There are two cases to consider: (a) p_j does not crash, or (b) p_j crashes, eventually recovers and remains always-up. For case (a), by the fair loss properties of the channels, p_j receives and then s-receives m: a contradiction. For case (b), since process p_i keeps on sending m to p_j, there is a time after which p_i sends m to p_j and none of them crash afterwards. As for case (a), by the fair loss property of the channels, p_j eventually receives m, then s-receives m: a contradiction. □

3 Broadcast Specifications

Informally, a *reliable broadcast* primitive ensures three properties [15]: (*validity*) every message broadcast by a correct process is delivered by the process; (*agreement*) processes agree on the set of messages they deliver; and (*integrity*) messages are not delivered more than once and cannot be delivered out of thin air. Roughly speaking, a *total order broadcast* is a reliable broadcast which also ensures the following property: (*total order*) processes deliver messages in the same order.

3.1 Reliable Broadcast

In a traditional crash-stop model [15], reliable broadcast was more precisely defined through two distinct primitives *broadcast* and *deliver* that satisfy the following properties:

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

Agreement: If a correct process delivers a message m, then every correct process eventually delivers m.

Integrity: For any message m, every correct process delivers m at most once, and only if m was previously broadcast by *sender(m)*.

Transposing these properties in a crash-recovery model can be done in various ways. Indeed, one could obtain several meaningful properties according to whether or not we consider the behaviour of processes that crash (and possibly recover), and whether or not we consider the behaviour of faulty processes - those which crash and do not recover, or keep crashing and recovering. In the following, we consider each property of reliable broadcast separately, and we give three meaningful variants of these properties in a crash-recovery model.[5] We first present three variants of these properties:

V.1. *Validity*: If a process p_i broadcasts a message m and then does not crash, p_i eventually delivers m.

V.2. *Uniform Validity*: If a correct process p_i broadcasts a message m, then p_i eventually delivers m.

V.3. *Strongly Uniform Validity*: If a process p_i broadcasts a message m, then p_i eventually delivers m.[6]

[5] We have considered properties that we believe are meaningful. We do not aim at being exhaustive here.

[6] It is easy to see that property V.3 is impossible to implement. In fact, V.3 would be impossible to implement

A.1. *Agreement*: If a process p_i delivers a message m and then does not crash, then any process that does not crash after p_i delivers m eventually delivers m.

A.2. *Uniform Agreement*: If a correct process p_i delivers a message m, then every correct process eventually delivers m.

A.3. *Strongly Uniform Agreement*: If a process delivers a message m, then every correct process eventually delivers m.

I.1. *Integrity*: For any message m, every process p_i that delivers m and then does not crash, delivers m at most once, and only if m was previously broadcast by *sender(m)*.

I.2. *Uniform Integrity*: For any message m, every correct process p_i delivers m at most once, and only if m was previously broadcast by *sender(m)*.

I.3. *Strongly Uniform Integrity*: For any message m, every process p_i delivers m at most once, and only if m was previously broadcast by *sender(m)*.

Combination. By combining one variant of each of these three kinds of properties, we obtain a specific form of reliable broadcast specification in a crash-recovery model. The *reliable broadcast* primitive defined with properties *V.1, A.1,* and *I.1* is the weakest among those specifications. We define the *uniform reliable broadcast* primitive with properties *V.2, A.2,* and *I.2*, and the *strongly uniform reliable broadcast* with properties *V.2, A.3,* and *I.3*. It makes also some sense to combine properties of different kinds. For instance, one could define the *weakly uniform reliable broadcast* by combining properties *V.1, A.2,* and *I.2*. This specification can be interesting in the context of replication. If the client crashes, then it is not necessary for the replicas to deliver the request and send back a reply, unless the client recovers and broadcasts again its request.

3.2 Total Order Broadcast

Total order broadcast is a primitive that requires processes to deliver the messages in the same order. This guarantee ensures that every correct process has the same *view* of the system. More precisely, a total order broadcast primitive ensures validity, agreement and integrity, plus the following property:

Total order: Let m and m' be any two messages. Let p_i and p_j be any two processes that deliver m. If p_i delivers m' before m then p_j also delivers m' before m.

As for reliable broadcast, defining a total order property in a crash-recovery model can be done

even if we weaken it to: *If a process p_i broadcasts a message m, then some correct process eventually delivers m.*

in various ways. In the following, we give three meaningful variants of the total order property in a crash-recovery model:

TO.1. *Total Order*: Let p_i and p_j be any two processes that deliver some message m. If p_i delivers some message m' before m and then does not crash, then if p_j also delivers m' and then does not crash, p_j delivers m' before m.

TO.2. *Uniform Total Order*: Let m and m' be any two messages. Let p_i and p_j be any two correct processes that deliver m. If p_i delivers m' before m then p_j also delivers m' before m.

TO.3. *Strongly Uniform Total Order*: Let m and m' be any two messages. Let p_i and p_j be any two processes that deliver m. If p_i delivers m' before m then p_j also delivers m' before m.

We combine these three properties with the precedent reliable broadcast properties and obtain different forms of total order broadcast. A *total order broadcast* primitive is defined with properties *V.1, A.1, I.1* and *TO.1*, which is our weakest specification of total order broadcast for the crash-recovery model. *Uniform total order broadcast* is defined with properties *V.2, A.2, I.2* and *TO.2*. Strongly uniform reliable broadcast is defined with properties *V.2, A.3, I.3* and *TO.3*, while *weakly uniform total order broadcast* is defined with properties *V.1, A.2, I.2* and *TO.2*.

3.3 Relationships

Before discussing the implementability of these specifications, we point out some preliminary results and relationships among our properties. We show that properties I.2 and I.3 are actually similar, and so are properties TO.2 and TO.3.

Proposition 2. *No algorithm can satisfy I.2 without satisfying I.3.*

Proof (sketch). Suppose by contradiction that an algorithm has a run r that satisfies I.2 but not I.3. This means that there is a process p_i and a time t at which either p_i delivers a message m that was never broadcast or p_i delivers m twice. One can obviously build a run r' similar to r until time t, and after time t, p_i recovers and never crashes again: contradicting I.2. □

Proposition 3. *No algorithm can satisfy TO.2 without satisfying TO.3.*

Proof (sketch). Suppose by contradiction that an algorithm has a run r that satisfies TO.2 but not TO.3. This means that there are two processes p_i and p_j, and a time t at which p_i delivers m before m' and p_j delivers m' without delivering m. One can obviously build a run r' similar to r until time t, and after time t, p_i and p_j recover and never crash again: contradicting

TO.2. □

4 Broadcast Transformations

This section first gives an overview of our notion of transformer algorithm, and then focuses on specific reliable broadcast and total order broadcast transformers. Correctness proofs of our total order broadcast transformers can be found in optional Appendix A (these are very similar to the correctness proofs of our reliable broadcast transformers).

4.1 Overview

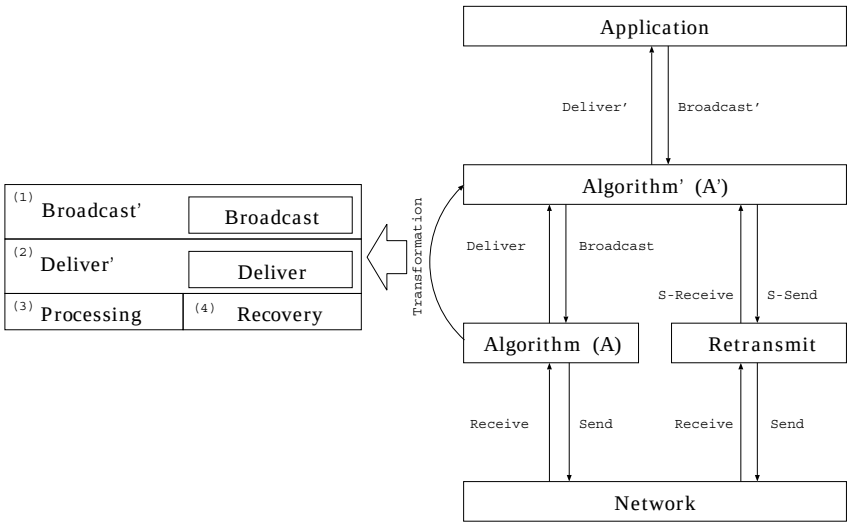


Figure 2: Transformation architecture

Let S and S' be any two broadcast specifications in the set $\{RB,\ WURB,\ URB,\ SURB,\ TOB,\ WUTOB,\ UTOB,\ SUTOB\}$, i.e., Reliable Broadcast, Weakly Uniform Reliable Broadcast, Uniform Reliable Broadcast, Strongly Uniform Reliable Broadcast, Total Order Broadcast, Weakly Uniform Total Order Broadcast, Uniform Total Order Broadcast, Strongly Uniform Total Order Broadcast. Assume S and S' are specifications of the same sort (i.e., reliable or total order) and assume S' is a stronger specification than S, e.g., if S is RB, then S' is $WURB$, URB or $SURB$.

A transformer $T_{S \rightarrow S'}$ is an algorithm that transforms any broadcast algorithm that implements S into a broadcast algorithm that implements S'. We denote by A the initial broadcast algorithm that implements S (associated with primitives *Broadcast* and *Deliver*), and A' the broadcast algorithm that implements S', resulting from the transformation (associated with

primitives *Broadcast'* and *Deliver'*). Figure 2 describes the architecture and the interaction between the layers that we consider in a transformer $T_{S \rightarrow S'}$. As described in the left part of Figure 2, a transformer is made up of four parts:

1. A *Broadcast'* primitive (R'-Broadcast, resp. TO'-Broadcast) based on the original *Broadcast* primitive (R-Broadcast, resp. TO-Broadcast)
2. A *Deliver'* primitive (R'-Deliver, resp. TO'-Deliver) based on the original *Deliver* primitive (R-Deliver, resp. TO-Deliver)
3. A *processing* procedure that is invoked when a process *Delivers* a message and before it *Delivers'* the message.
4. A *recovery* procedure that is invoked when a process recovers from a crash. Each transformer has in addition an initialisation procedure that initialises its variables.

A transformer for reliable broadcast is similar to the corresponding transformer for total order broadcast except that specific *processing* and *recovery* procedures are plugged in. We present in the following subsections the transformers: $T_{RB \rightarrow URB}$, $T_{URB \rightarrow SURB}$, $T_{TOB \rightarrow UTOB}$ and $T_{UTOB \rightarrow SUTOB}$.[7]

4.2 Reliable Broadcast

We describe here the transformers $T_{RB \rightarrow URB}$ and $T_{URB \rightarrow SURB}$, and we state and prove their correctness. We say that a process p_i R'-Broadcasts a message m once p_i returns from the invocation of R'-Broadcast. As in [1], we say that a process p_i R'-Delivers a message m when p_i stores m into an adequate stable storage location. The primitive R-Deliver is implemented as a callback and we make the assumption that when R-Delivering a message m, the algorithm A stores m into an adequate stable storage location.

Transformer $T_{RB \rightarrow URB}$. The algorithm $T_{RB \rightarrow URB}$ is presented in Figure 4, it works as follows for a given process p_i. First, to ensure property V.2, p_i stores all messages that it R-Broadcast, in case p_i crashes and recovers. Process p_i ensures property I.2 by storing the messages that are R'-Delivered into stable storage (in order not to R'-Deliver them twice). To satisfy property A.2, p_i s-sends to all processes the messages that are R-Delivered. To illustrate the need for this forwarding phase, consider the case depicted in Figure 3: process p_1 R'-Broadcasts m, R'-Delivers

[7] Due to a lack of space, we do not present $T_{RB \rightarrow WURB}$, $T_{WURB \rightarrow URB}$, $T_{TOB \rightarrow WUTOB}$ and $T_{WUTOB \rightarrow UTOB}$.

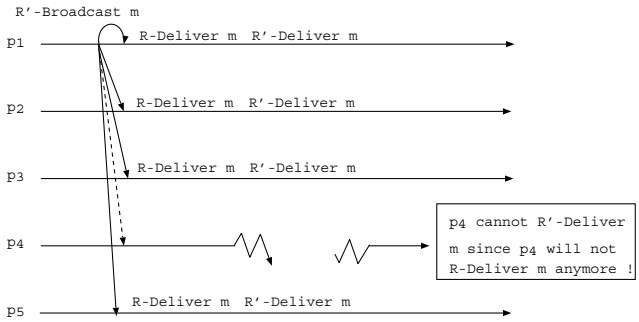


Figure 3: Uniform agreement is violated

m and then does not crash, while process p_4 does not R-Deliver m, then crashes and recovers. Process p_4 will never R'-Deliver m since p_4 will never R-Deliver m (with the specification of reliable broadcast, once a process crashes, it does not have to R-Deliver m). Therefore, some process, e.g., p_1, has to s-send m to every process. Finally, the recovery procedure is invoked when p_i recovers from a crash. The recovery procedure is composed of the following three phases: (i) p_i R-Broadcasts again the messages that were R'-Broadcast in order to ensure property V.2, (ii) p_i R'-Delivers all messages that were R-Delivered but not R'-Delivered, and (iii) p_i s-sends to every process the messages that were R'-Delivered to ensure property A.2. Phase (ii) occurs when p_i R-Delivers some message m and then crashes before R'-Delivering m. Since p_i stores the messages that it R-Delivered into stable storage, p_i can retrieve theses messages when it recovers. Note that for the sake of modularity, the code that a process executes before R'-Delivering has been factored out in the processing primitive: we will see in the next subsections that this factorisation helps having generic transformers.

Lemma 4. *The algorithm of Figure 4 transforms the property V.1 of A into the property V.2 of A': If a correct process p_i R'-Broadcasts a message m, then p_i eventually R'-Delivers m.*

Proof. Let p_i be a correct process that R'-Broadcasts m and assume by contradiction that p_i never R'-Delivers m (i.e., p_i violates property V.2). There are two cases to consider: (i) p_i does not crash, and (ii) p_i crashes, recovers and remains always-up. For case (i), since p_i does not crash, then by the property V.1 of A, p_i eventually R-Delivers m (line 13), then R'-Delivers m: a contradiction. For case (ii), if p_i crashes and recovers, there is a time after which p_i stops crashing and remains always-up. When p_i recovers, p_i retrieves either (a) the messages it R-Delivered and never R'-Delivered, or (b) the messages it R'-Broadcast (including m) at line 18. For case (a), p_i R'-Delivers m at lines 20-21: a contradiction. For case (b), p_i R-Broadcasts these recovered messages (including m) at line 19. We are certain that p_i has stored m into stable storage at line 11 since a process has R'-Broadcast m only when it returns from the invocation

```
1: for each process p_i:
2: procedure initialisation:
3:    msgSent[] ← ⊥; r'_delivered[] ← ⊥
4: procedure processing(m)
5:    if m ∉ r'_delivered then
6:       r'_delivered ← r'_delivered ∪ m
7:       store{r'_delivered}                                   {R'-Deliver(m)}
8: procedure R'-Broadcast(m)
9:    if m ∉ msgSent then
10:      msgSent ← msgSent ∪ m
11:      store msgSent
12:   R-Broadcast(m)
13: upon R-Deliver(m) do
14:   s-send(m) to all; processing(m)
15: upon s-receive m from p_j do
16:   processing(m)
17: upon recovery do
18:   initialisation; retrieve{msgSent, r'_delivered, r_delivered}
19:   R-Broadcast(msgSent)
20:   for all m' ∈ r_delivered do
21:      processing(m');
22:   s-send(r'_delivered) to all
```

Figure 4: $\mathrm{T}_{RB \rightarrow URB}$

of R'-Broadcast. By line 11 of the algorithm, we ensure that the forced log will be executed before returning from the invocation of R'-Broadcast. Since p_i never R-Delivered m, then by the property V.1 of A, p_i R-Delivers m at line 13 and then R'-Delivers m: a contradiction. □

Lemma 5. *The algorithm of Figure 4 transforms the property A.1 of A into the property A.2 of A′: If a correct process R'-Delivers a message m, then every correct process eventually R'-Delivers m.*

Proof. Let p_i be a correct process that R'-Delivers m and assume by contradiction that there is a correct process p_j that does not R'-Deliver m. There are four cases to consider: (a) p_i and p_j do not crash, or (b) p_i crashes, recovers and remains always-up, and p_j does not crash, or (c) p_i does not crash, and p_j crashes, recovers and remains always-up, and finally (d) both processes p_i and p_j crash, recover and remain always-up. For case (a), since both processes do not crash, by the property A.1 of A, p_j R-Delivers m, therefore p_j R'-Delivers m: a contradiction. For case (b), since p_i is correct, there is a time after which p_i stops crashing and remains always-up. After recovering, p_i retrieves the messages that it R'-Delivered before, and s-sends them to every process at line 22. By the validity property of the retransmission module, p_j eventually s-receives m and R'-Delivers m at line 7: a contradiction. For case (c), since p_j is correct, there is a time after which p_j stops crashing and remains always-up. Before R'-Delivering m, p_i s-sends m to p_j at line 14. As for case (b), p_j then eventually s-receives m and R'-Delivers m: a contradiction. Finally, case (d) is a mix of case (b) and (c); there is a time after which both processes p_i and

p_j stop crashing and remain always-up. As for case (b) and (c), p_i s-sends m to p_j at line 14 or 22, p_j then eventually s-receives m and R'-Delivers m: a contradiction. □

Lemma 6. *The algorithm of Figure 4 transforms the property I.1 of A into the property I.2 of A′: For any message m, every correct process p_i R'-Delivers m at most once, and only if m was previously R'-Broadcast by sender(m).*

Proof. For the first part of the property, suppose by contradiction that a correct process p_i R'-Delivers m more than once. We have two cases to consider: (i) p_i does not crash, or (ii) p_i crashes, recovers and remains always-up. For case (i), this is clearly impossible, since before R'-Delivering m, p_i appends m to the set *r'_delivered* at line 6, and checks for m in the set *r'_delivered* at guard line 5: a contradiction. For case (ii), there is a time after which p_i stops crashing and remains always-up. Remember that when R'-Delivering m, p_i stores the set *r'_delivered* into stable storage at line 7. When p_i recovers, it retrieves the set *r'_delivered* and therefore cannot go through guard line 5 twice since m will be in the set *r'_delivered*: a contradiction. The second part follows from the no creation property of the channels. This property prevents the case of s-receiving (resp. R-Delivering) messages that were not s-sent (resp. R-Broadcast). □

Proposition 7. *The algorithm of Figure 4 transforms a reliable broadcast into a uniform reliable broadcast.*

Proof. Follows directly from lemmata 4, 5, and 6. □

Transformer $T_{URB \to SURB}$. For the rest of this section, we assume that there is a majority of correct processes in the system. Figure 5 presents transformer $T_{URB \to SURB}$ that assumes a uniform reliable broadcast as the lower building block (which could be implemented by transformer $T_{RB \to URB}$ on top of reliable broadcast). The algorithm works as follows for a given process p_i. Once a process p_i R-Delivers a message m, p_i s-sends an acknowledgement of m (ACK(m)) to every process. Every process waits to s-receive a majority of ACK(m) before R'-Delivering m. As for $T_{RB \to URB}$, when p_i recovers, p_i R-Broadcasts again all messages it R'-Broadcast previously in case p_i was not able to invoke the R-Broadcast primitive before crashing; this allows every correct process to R-Deliver and to acknowledge every message. In the recovery procedure, p_i s-sends acknowledgements of all R-Delivered messages to all processes in case a process would wait for an acknowledgement of m before R'-Delivering m. In contrast to $T_{RB \to URB}$, when p_i recovers, p_i cannot directly R'-Deliver the messages that it R-Delivered but did not R'-Deliver,

this would violate property A.3. Thanks to our modular approach, the very same processing procedure than in $T_{RB \rightarrow URB}$ can be reused here.

```
1: for each process p_i:
2: procedure initialisation:
3:    ackReceived[] ← ⊥; r'_delivered[] ← ⊥
4: procedure processing(m)
5:    if m ∉ r'_delivered then
6:       r'_delivered ← r'_delivered ∪ m
7:       store{r'_delivered}                                   {R'-Deliver(m)}
8: procedure R'-Broadcast(m)
9:    R-Broadcast(m)
10: upon R-Deliver(m) do
11:    s-send(ACK(m)) to all
12: upon s-receive ACK(m) from p_j do
13:    if p_j ∉ ackReceived[m] then
14:       ackReceived[m] ← ackReceived[m] ∪ p_j
15:       if ackReceived[m].size() > ⌈(n+1)/2⌉ and m ∉ r'_delivered then
16:          processing(m)
17: upon recovery do
18:    initialisation; retrieve{r_delivered, r'_delivered}
19:    for all m' ∈ r_delivered do
20:       s-send(ACK(m')) to all
```

Figure 5: $T_{URB \rightarrow SURB}$

Lemma 8. *The algorithm of Figure 5 preserves the property V.2 of A into A′: If a correct process p_i R'-Broadcasts a message m, then p_i eventually R'-Delivers m.*

Proof. Let p_i be any correct process that R'-Broadcasts m and assume by contradiction that p_i never R'-Delivers m (i.e., p_i violates property V.2). There are two cases to consider: (i) p_i does not crash, or (ii) p_i crashes, recovers and remains always-up. For case (i), since (a) p_i R-Broadcasts m and waits for a majority of ACK(m), (b) there is a majority of correct processes, (c) there is a time after which those correct processes stop crashing and remain always-up, and (d) by the property A.2 of A, eventually every correct process R-Delivers m and then acknowledges m by s-sending ACK(m). By the validity property of the retransmission module, p_i then s-receives a majority of ACK(m) and R'-Delivers m: a contradiction. For case (ii), there is a time after which p_i stops crashing and remains always-up, then (a) p_i retrieves and s-sends, when recovering, the messages it R-Delivered but did not R'-Deliver, and (b) for the same reasons invoked for case (i), p_i eventually s-receives a majority of ACK(m) and therefore R'-Delivers m: a contradiction. □

Lemma 9. *The algorithm of Figure 5 transforms the property A.2 of A into the property A.3 of A′: If a process R'-Delivers a message m, then every correct process eventually R'-Delivers m.*

Proof. Let p_i be any process that R'-Delivers a message m and assume by contradiction that a

correct process p_j does not R'-Deliver m. When p_i R'-Delivers m, p_i has to s-receive a majority of ACK(m) at line 15. Since we assume a majority of correct processes, there is at least one correct process p_j that has R-Delivered m since ACK(m) messages are only s-sent when a process has R-Delivered m (line 11). There is a time after which every correct process stops crashing and remains always-up. Since p_j R-Delivered m, by the property A.2 of A, every correct process R-Delivers m and then acknowledges m when recovering. Hence, p_j s-receives a majority of ACK(m) and R'-Delivers m: a contradiction. □

Lemma 10. *The algorithm of Figure 5 preserves the property I.2 of A into A′: For any message m, every correct process p_i R'-Delivers m at most once, and only if m was previously R'-Broadcast by sender(m).*

Proof. Identical to the proof of lemma 6. □

Proposition 11. *The algorithm of Figure 5 transforms a uniform reliable broadcast into a strongly uniform reliable broadcast.*

Proof. Follows directly from lemmata 8, 9 and 10. □

4.3 Total Order Broadcast

We present here total order broadcast transformers $T_{TOB \rightarrow UTOB}$ and $T_{UTOB \rightarrow SUTOB}$. As shown in Figure 6, the structure of the total order transformers presented in this subsection is identical to those of reliable broadcast. The only differences are (i) the underlying primitive invoked, (ii) the modified processing and recovery procedures, and (iii) variable renaming, e.g., the *r'_delivered* variable is replaced by the k^{th} batch of messages *to'_delivered*. Otherwise, the transformer algorithms remain exactly the same. We say that a process p_i TO'-Broadcasts a message m once p_i returns from the invocation of TO'-Broadcast. We say that p_i TO'-Delivers m when p_i performs a forced log of m into an adequate stable storage location.[8] We assume that there is a deterministic rule by which p_i TO-Delivers or TO'-Delivers a batch of messages. As for reliable broadcast, the primitive TO-Deliver is implemented as a callback and we assume that when TO-Delivering, the algorithm A stores the batch of messages into an adequate stable storage location.

Transformer $T_{TOB \rightarrow UTOB}$. The structure of the transformer $T_{TOB \rightarrow UTOB}$ presented in Figure 7 is the same as the one of $T_{RB \rightarrow URB}$. The processing procedure ensures the total order

[8] Message m is part of the k^{th} batch of messages *to'_delivered*

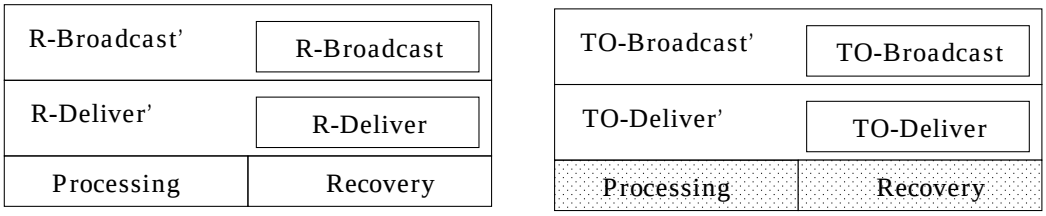


Figure 6: Differences (in shade) between transformers

property: when a process p_i TO-Delivers or s-receives the k^{th} batch of messages, p_i either (a) discards the batch of messages if k is lower than the one expected ($nextBatch$) [$k < nextBatch$], or (b) puts the batch of messages into the set *awaitingToBeDelivered* if k is greater than the one expected (lines 5-6) [$k > nextBatch$], or finally (c) if the batch of messages received is the one that was expected [$k = nextBatch$], p_i TO'-Delivers it (line 10), and increments the value of the next batch of messages that is expected (line 11). Process p_i then verifies if there are other batches of messages that can be TO'-Delivered (lines 12-15).

```
1: for each process p_i:
2: procedure initialisation:
3:    to'_delivered[] ← ⊥; awaitingToBeDelivered[] ← ⊥; msgSet[] ← ⊥; nextBatch ← 1; msgSent[] ← ⊥
4: procedure processing(l, msgSet)
5:    if l > nextBatch then
6:       awaitingToBeDelivered[l] = msgSet
7:    else if l = nextBatch and to'_delivered[nextBatch] = ⊥ then
8:       to'_delivered[nextBatch] ← msgSet - to'_delivered
9:       atomically deliver all messages in to'_delivered(nextBatch) in some deterministic order
10:      store{to'_delivered,nextBatch}                                   {TO'-Deliver}
11:      nextBatch ← nextBatch+1
12:      while awaitingToBeDelivered[nextBatch] ≠ ⊥ do
13:         to'_delivered[nextBatch] ← awaitingToBeDelivered[nextBatch] - to'_delivered
14:         atomically deliver all messages in to'_delivered(nextBatch) in some deterministic order
15:         store{to'_delivered,nextBatch}; nextBatch ← nextBatch+1       {TO'-Deliver}
16: procedure TO'-Broadcast(m)
17:    if m ∉ msgSent then
18:       msgSent ← msgSent ∪ m
19:       store msgSent
20:    TO-Broadcast(m)
21: upon TO-Deliver(k,to_delivered[k]) do
22:    s-send(k,to_delivered[k]) to all; processing(k,to_delivered[k])
23: upon s-receive (k,to_delivered[k]) from p_j do
24:    processing(k,to_delivered[k])
25: upon recovery do
26:    initialisation; retrieve{msgSent,to_delivered,to'_delivered,nextBatch}
27:    nextBatch ← nextBatch+1
28:    TO-Broadcast(msgSent)
29:    for all l ∈ to_delivered do
30:       processing(l,to_delivered[l])
31:    for all l' ∈ to'_delivered do
32:       s-send(l',to'_delivered[l']) to all
```

Figure 7: $T_{TOB \rightarrow UTOB}$

Transformer $T_{UTOB \rightarrow SUTOB}$**.** The structure of transformer $T_{UTOB \rightarrow SUTOB}$ presented in Figure 8 is similar to the one of $T_{URB \rightarrow SURB}$. Once a process p_i TO-Delivers a message m, p_i

s-sends an acknowledgement for the k^{th} batch that contains m to every process (ACK(k)). Every process waits for a majority of ACK(k) before TO'-Delivering a message. The processing procedure ensures property TO.3 and is identical to the one of $T_{TOB \rightarrow UTOB}$. The recovery procedure is a mix of the ones of $T_{URB \rightarrow SURB}$ and $T_{TOB \rightarrow UTOB}$. When recovering, p_i retrieves the messages it TO-Delivered and TO'-Delivered, and updates the value of the next expected batch. Process p_i then s-sends ACK(k) for all TO-Delivered messages to all processes.

```
1: for each process p_i:
2: procedure initialisation:
3:    to'_delivered[] ← ⊥; nextBatch ← 1; awaitingToBeDelivered[] ← ⊥; ackReceived[] ← ⊥
4: procedure processing(l, msgSet)
5:    if l > nextBatch then
6:       awaitingToBeDelivered[l] = msgSet
7:    else if l = nextBatch and to'_delivered[nextBatch] = ⊥ then
8:       to'_delivered[nextBatch] ← msgSet - to'_delivered
9:       atomically deliver all messages in to'_delivered(nextBatch) in some deterministic order
10:      store{to'_delivered,nextBatch}                                   {TO'-Deliver}
11:      nextBatch ← nextBatch+1
12:      while awaitingToBeDelivered[nextBatch] ≠ ⊥ do
13:         to'_delivered[nextBatch] ← awaitingToBeDelivered[nextBatch] - to'_delivered
14:         atomically deliver all messages in to'_delivered(nextBatch) in some deterministic order
15:         store{to'_delivered,nextBatch}; nextBatch ← nextBatch+1       {TO'-Deliver}
16: procedure TO'-Broadcast(m)
17:    TO-Broadcast(m)
18: upon TO-Deliver(k,to_delivered[k]) do
19:    s-send(ACK(k))) to all
20: upon s-receive ACK(k) from p_j do
21:    if p_j ∉ ackReceived[k] then
22:       ackReceived[k] ← ackReceived[k] ∪ p_j
23:       if ackReceived[k].size() > ⌈(n+1)/2⌉ then
24:          processing(k,to_delivered[k])
25: upon recovery do
26:    initialisation; retrieve{to_delivered, to'_delivered, nextBatch}
27:    nextBatch ← nextBatch+1
28:    for all l ∈ to_delivered do
29:       s-send(ACK(l)) to all
```

Figure 8: $T_{UTOB \rightarrow SUTOB}$

5 Algorithms: Optimisations and Lower Bounds

In the previous section, we did not consider specific algorithms but we have shown how to transform *any* given algorithm that satisfies a given specification into an algorithm that satisfies a stronger specification. We focus here on specific algorithms. We show how we build crash-recovery resilient broadcast algorithms based on actual algorithms from the crash-stop model (namely, the algorithms of [15] and [7]) and our transformers. We then show how to optimise these algorithms in terms of forced logs and messages using a systematic approach. We then describe for each type of algorithm its analytical performance in terms of forced logs and communication steps, and then we give some interesting lower bounds. Finally, we give

some experimental performance results of the implementation of those algorithms in a local area network.

5.1 Reliable Broadcast

We first show that the crash-stop reliable broadcast algorithm from [15], when adapted to fair lossy channels, satisfies the properties V.1, A.1, I.1 of reliable broadcast. This algorithm can hence be used as a building block to devise stronger reliable broadcast algorithms, e.g., a crash-recovery uniform reliable broadcast algorithm. Figure 9 presents the reliable broadcast algorithm of [15] adapted to fair lossy channels, i.e, we basically replace the send (resp. receive) primitive by the s-send (resp. s-receive) primitive of the retransmission module.

```
1: for each process p_i:
2: procedure initialisation:
3:    r_delivered[] ← ⊥
4: procedure processing(m)
5:    if m ∉ r_delivered then
6:       s-send m to all \p_i
7:       r_delivered ← r_delivered ∪ m                    {R-Deliver(m)}
8: procedure R-Broadcast(m)
9:    s-send(m) to p_i
10: upon s-receive m from p_j do
11:    processing(m)
12: upon recovery do
13:    initialisation
```

Figure 9: Adaptation of the reliable broadcast of [15]

Lemma 12. *The algorithm of Figure 9 satisfies the property V.1 of reliable broadcast: If a process p_i R-Broadcasts a message m and then does not crash, p_i eventually R-Delivers m.*

Proof. Suppose by contradiction that p_i R-Broadcasts a message m, then does not crash and never R-Delivers m. By the algorithm of Figure 9 and the validity property of the retransmission module, p_i s-sends m, s-receives m and then R-Delivers m: a contradiction. □

Lemma 13. *The algorithm of Figure 9 satisfies the property A.1 of reliable broadcast: If a process p_i R-Delivers a message m and then does not crash, then any process that does not crash after p_i R-Delivers m eventually R-Delivers m.*

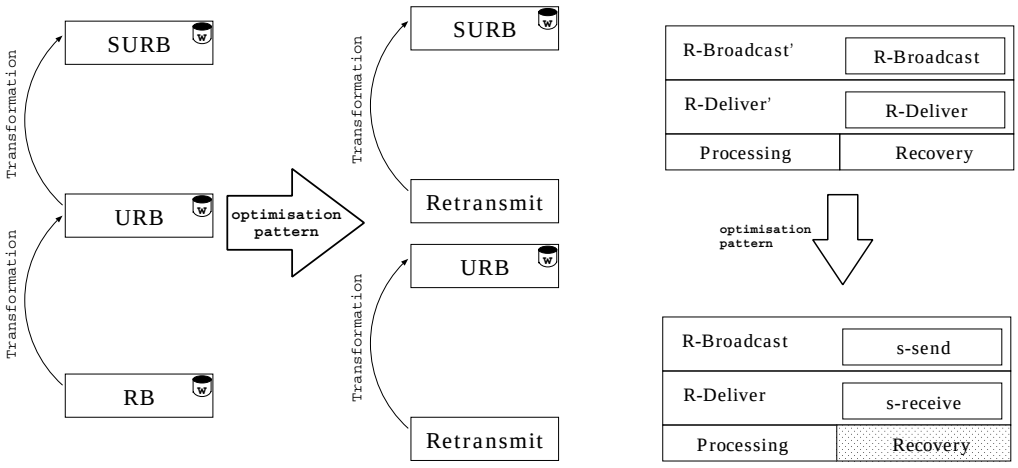
Proof. Let p_i be a process that R-Delivers m and then does not crash, let p_j be a process that does not crash after p_i R-Delivers m, and assume by contradiction that p_j never R-Delivers m. When p_i R-Delivers m, p_i s-sends m to every process except itself, by the validity property of the retransmission module, p_j s-receives m and then R-Delivers m: a contradiction. □

Lemma 14. *The algorithm of Figure 9 satisfies the property I.1 of reliable broadcast: For any message m, every process p_i that R-Delivers m and then does not crash, R-Delivers m at most once, and only if m was previously R-Broadcast by sender(m).*

Proof. Assume that a process p_i R-Delivers a message m and then does not crash. By the no creation property of the channels, p_i cannot s-receive a message out of thin air, and since p_i does not crash after R-Delivering m, the guard line 5 prevents p_i from R-Delivering m twice. □

Proposition 15: *The algorithm of Figure 9 satisfies the properties V.1, A.1, and I.1 of reliable broadcast.*

Proof. Follows directly from lemmata 12, 13 and 14. □



(a) Merging layers

(b) Replacing primitives

Figure 10: Optimisation pattern for reliable broadcast

Uniform Reliable Broadcast. When applying transformer $T_{RB \rightarrow URB}$ to the reliable broadcast of Figure 9, we obtain a uniform reliable broadcast algorithm (V.2, A.2, I.2). Our transformation introduces however, (i) some redundant forced logs, and (ii) additional messages. Indeed, in $T_{RB \rightarrow URB}$ (see Figure 4), the variable *r'_delivered* is redundant with the variable *r_delivered*. One of these forced logs is actually useless and can be eliminated if both layers (A and A') are merged. In fact, when merging all layers into one, (i) numerous forced logs can be removed, and (ii) numerous messages can be saved: both using a systematic approach. Intuitively, as shown in Figure 10, the optimisation pattern is the following:

- The middle layer (e.g., layer A) is removed.

- The R-Broadcast (resp. R-Deliver) primitive is replaced by the s-send (resp. s-receive) primitive.
- Thanks to our modular approach, the processing and recovery procedures from the transformers of Section 4.2 can be reused for these algorithms.

Figure 11 gives an optimised algorithm for uniform reliable broadcast. A close look at the code in Figure 11 shows that it is exactly the same as the one from Figure 4, except that the optimisation pattern has been applied, e.g., (a) the R-Broadcast primitive of Figure 4 is replaced with the s-send primitive in Figure 11, (b) the R'-Broadcast primitive of Figure 4 is replaced with the R-Broadcast primitive in Figure 11, and (c) the R-Deliver primitive disappears, since it now makes double usage with the s-receive primitive. Due to a lack of space and since the correctness proofs for this algorithm are similar to those of Figure 4, we omit them here.

```
1: for each process p_i:
2: procedure initialisation:
3:    msgSent[] ← ⊥; ur_delivered[] ← ⊥
4: procedure processing(m)
5:    if m ∉ ur_delivered then
6:       ur_delivered ← ur_delivered ∪ m
7:       store{ur_delivered}                                   {UR-Deliver(m)}
8: procedure UR-Broadcast(m)
9:    if m ∉ msgSent then
10:      msgSent ← msgSent ∪ m
11:      store{msgSent}
12:   s-send(m) to all
13: upon s-receive m from p_j do
14:   processing(m)
15: upon recovery do
16:   initialisation; retrieve{msgSent, ur_delivered}
17:   s-send(ur_delivered); s-send(msgSent)
```

Figure 11: Optimised uniform reliable broadcast (V.2, A.2, I.2)

Strongly Uniform Reliable Broadcast. Applying transformer $T_{URB \rightarrow SURB}$ on the precedent uniform reliable broadcast enables us to obtain a strongly uniform reliable broadcast (V.2, A.3, I.3). However the resulting algorithm contains some redundant forced logs. We use the same optimisation pattern applied to $T_{URB \rightarrow SURB}$ and we obtain the optimised algorithm of Figure 12. Note that $T_{RB \rightarrow URB}$ adds some messages, while $T_{URB \rightarrow SURB}$ does not. However, there is an added forced log compared to transformer $T_{URB \rightarrow SURB}$: once a message m has been s-received, m is stored on stable storage. This forced log is mandatory since the optimised algorithm cannot rely anymore on the properties of uniform reliable broadcast but only on those of the retransmission module. If this forced log was not performed, the optimised algorithm would violate property A.3. Due to a lack of space and since the correctness proofs are similar to those

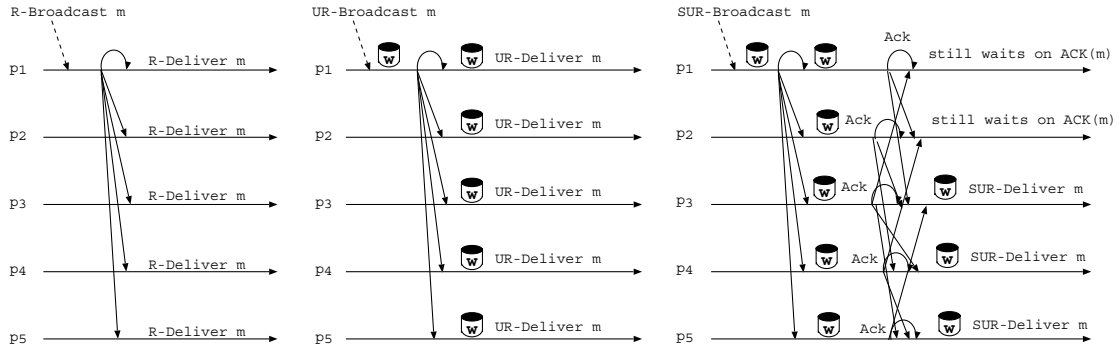
of $T_{URB \rightarrow SURB}$, we omit them here.

```
1: for each process p_i:
2: procedure initialisation:
3:    msgReceived[] ← ⊥; msgSent[] ← ⊥; sur_delivered[] ← ⊥; ackReceived[] ← ⊥
4: procedure processing(m)
5:    if m ∉ sur_delivered then
6:       sur_delivered ← sur_delivered ∪ m
7:       store{sur_delivered}                                          {SUR-Deliver(m)}
8: procedure SUR-Broadcast(m)
9:    if m ∉ msgSent then
10:      msgSent ← msgSent ∪ m
11:      store{msgSent}
12:   s-send(m)
13: upon s-receive m from p_j do
14:   if m = ACK(m) and p_j ∉ ackReceived[m] then
15:      ackReceived[m] ← ackReceived[m] ∪ p_j
16:      if ackReceived[m].size() > ⌈(n+1)/2⌉ and m ∉ sur_delivered then
17:         processing(m)
18:   else if m ≠ ACK(m) and m ∉ msgReceived then
19:      msgReceived ← msgReceived ∪ m; store{msgReceived}
20:      s-send(ACK(m)); s-send(m)
21: upon recovery do
22:   initialisation; retrieve{msgSent, msgReceived sur_delivered}
23:   s-send(msgSent); s-send(msgReceived); s-send(sur_delivered)
24:   for all m' ∈ msgReceived do
25:      s-send(ACK(m'))
```

Figure 12: Optimised strongly uniform reliable broadcast (V.2, A.3, I.3)

Analytical Performance and Lower Bounds. Figure 13 depicts the communication and stable storage pattern of several reliable broadcast algorithms: (a) the reliable broadcast of [15], (b) the uniform reliable broadcast of Figure 11, and (c) the strongly uniform reliable broadcast of Figure 12. Figure 13 considers nice runs, i.e., no process or link crashes. Figure 13(a) shows that the reliable broadcast algorithm does not perform any forced logs. However, the uniform reliable broadcast algorithm performs one forced log when UR-Broadcasting a message m, and one forced log when UR-Delivering m. Finally, Figure 13(c) shows that strongly uniform reliable broadcast has an additional forced log per process compared to uniform reliable broadcast, i.e., it performs a forced log when a process s-receives a message m for the first time. To ease reading, we did not draw all acknowledgements on the diagram; actually each process s-sends acknowledgements to every other process. Figure 13(c) presents a scenario where the first two processes s-receive only two acknowledgements, and therefore cannot SUR-Deliver a message m. On the opposite, the last three processes s-receive a majority of acknowledgements and indeed SUR-Deliver m.

One can trivially verify that, in nice runs and for any given message m reliable broadcast and uniform reliable broadcast require one communication step (resp. n_c messages, where n_c is the number of correct processes in the system) before m is delivered; while strongly uniform



(a) Regular (b) Uniform (c) Strongly uniform

Figure 13: Communication and stable storage pattern for reliable broadcast

reliable broadcast needs two communication steps (resp. $n_c^2 + n_c$ messages) to deliver m. These bounds are clearly minimal. We show now that our reliable broadcast algorithms are minimal in the number of logs they perform. We state for uniform reliable broadcast that if a process p_i UR-Delivers a message m, then (i) p_i has performed at least one forced log, and (ii) at least two forced logs have been performed in the system. Intuitively, as depicted in Figure 13(b), when p_i UR-Delivers m, p_i must perform one forced log. However, the process that UR-Broadcasts must also perform another forced log; thus, at least two forced logs have been performed in the system.

Lemma 16. *Consider any uniform reliable broadcast algorithm A, A cannot satisfy the properties of uniform reliable broadcast if A does not perform at least one forced log.*

Proof. Assume by contradiction that there is a uniform reliable broadcast algorithm A that does not use stable storage. Let $R(m, G_{faulty}, G_{au})$ be the set of runs of A such that (1) the only message broadcast is m; (2) processes in G_{faulty} crash at the beginning and never recover; (3) processes in G_{au} never crash; and (4) processes not in $G_{au} \cup G_{faulty}$ crash at the beginning, recover afterwards and never crash again. We show now that there exists disjoint subsets of processes G and G' such that in some run $r \in R(m, G, G')$ some correct process UR-Delivers m more than once (which violates property A.2). In $R(m, G, G')$, we have faulty, always-up and eventually always-up processes. Let p_i be any correct process. Consider two runs r_0 and r_1, both belonging to $R(m, G, G')$. Since $r_0 \in R(m, G, G')$ and A solves uniform reliable broadcast, we can suppose that in r_0, p_i UR-Delivers m and then crashes. When p_i recovers, p_i begins with all its values initialised as it would initially at time 0; we say that p_i is in state *initialised*. Now consider run r_1 where p_i does not UR-Deliver m and crashes. When p_i recovers, it is also in the

same state *initialised*. However, for run r_0 and r_1 to satisfy the properties A.2 and I.2, p_i must behave differently in both runs. In order not to violate property I.2, p_i **must not** UR-Deliver m in r_0; in order not to violate property A.2, p_i **must** UR-Deliver m in r_1: a contradiction. The action of UR-Deliver is local, and p_i in each run has the same state *initialised*. We have then two cases to consider for each run: (i) p_i UR-Delivers m, or (ii) p_i does not UR-Deliver m. A is deterministic and therefore if A UR-Delivers m in r_0, so will it in r_1 (and vice-versa). Cases (i) and (ii) are trivial, both violate some property, either I.2 is violated in run r_0 (since p_i UR-Delivers m twice), or A.2 is violated in run r_1 (since p_i does not UR-Deliver m at all).□

Lemma 17. *Consider any uniform reliable broadcast algorithm A, let p_i be any process, if p_i UR-Delivers a message then p_i has performed at least one forced log.*

Proof. Assume that UR-Deliver is a local event which is triggered once the message m has been logged into stable storage. Assume moreover that there is a uniform reliable broadcast algorithm A that uses stable storage. As for the proof of lemma 16, consider $R(m, G_{faulty}, G_{au})$ to be the set of runs of A. Let p_i be any correct process. Consider two runs r_0 and r_1. Since $r'_0 \in R(m, G_{faulty}, G_{au})$ and A solves uniform reliable broadcast, we can suppose that in r_0, p_i stores m (therefore UR-Delivers m) and then p_i crashes. When p_i recovers, p_i retrieves m and is in state *retrieved* (not *initialised*). Now consider run r_1 where p_i does not UR-Deliver m and crashes. When p_i recovers, it cannot retrieve any message from stable storage and thus is in state *initialised*. For run r_0 and r_1 to satisfy the properties A.2 and I.2, p_i has to behave differently in both runs. In order not to violate property I.2, p_i **must not** UR-Deliver m in r_0; and in order not to violate property A.2, p_i **must** UR-Deliver m in r_1. In contrary to the scenario of lemma 16, p_i is in a different state in r_0 and r_1. A is deterministic but since p_i has a different state in both runs, both processes do not need to execute the same steps. Consider run r_0, p_i will not UR-Deliver m since A is deterministic and solves uniform reliable broadcast; p_i must only UR-Deliver m at most once, otherwise it violates property I.2. Now consider run r_1, since A solves uniform reliable broadcast, p_i must UR-Deliver m otherwise it violates the uniform agreement property. Now if p_i crashes and recovers in both runs, p_i will have the same state *retrieved*. We have proved that a process has to perform at least one forced log per message and per correct process. Since we are not able to distinguish between always-up, faulty or eventually always-up processes, all processes need to perform at least one forced log even if they are always-up processes. □

Lemma 18. *Consider any uniform reliable broadcast algorithm A, let p_i be any process, if p_i UR-Delivers a message, then at least two forced logs have been performed in the system.*

Proof. With lemma 17, a correct process requires at least one forced log to UR-Deliver a message m. We first to show that one forced log is necessary before returning from the UR-Broadcast primitive and then show that this forced log is not the one accounted for lemma 17. Let p_i be any correct process. Consider four runs r_0, r_1, r_2 and r_3, all belonging to $R(m, G_{faulty}, G_{au})$. Consider r_0 where p_i UR-Broadcasts a message m and then crashes. Suppose moreover that no process (including p_i) receives m. When p_i recovers, p_i is in state *initialised* and no process has some knowledge of m; m will be lost forever. This behaviour violates property V.2. To overcome this case, we need to store m in stable storage before UR-Broadcasting m. Now, consider run r_1: p_i stores m in stable storage, UR-Broadcasts m and then crashes. When p_i recovers, it can retrieve message m and thus is not in state *initialised* but *retrieved*. Process p_i could keep on retransmitting m and therefore UR-Deliver m. This shows that one forced log is required before returning from the UR-Broadcast primitive, we show now that this forced log is not the one from lemma 17 and that a second forced log is necessary.

Consider run r_2 where p_i stores m, UR-Broadcasts m. Since $r_2 \in R(m, G_{faulty}, G_{au})$ and A solves uniform reliable broadcast, we can suppose that in r_2, p_i UR-Delivers m and then crashes. When p_i recovers, p_i retrieves m and is in state *retrieved*. Consider run r_3 where p_i stores m, UR-Broadcasts m, does not UR-Deliver m and then crashes. When p_i recovers, p_i retrieves also m and is in state *retrieved*. However, for run r_2 and r_3 to satisfy the properties A.2 and I.2, p_i must behave differently in both runs. In order not to violate property I.2, p_i **must not** UR-Deliver m in r_2; and in order not to violate the property A.2, p_i **must** UR-Deliver m in r_3: a contradiction. The action of UR-Deliver is local, and p_i in each run has the same state *retrieved*. We have then two cases for both runs: (i) p_i UR-Delivers m, or (ii) p_i does not UR-Deliver m. A is deterministic and therefore if p_i UR-delivers m in r_2, so will p_i in r_3 (and vice-versa). Cases (i) and (ii) are trivial, both violate some property (i) I.2 is violated in run r_2 (since p_i UR-Delivers m twice), and (ii) A.2 is violated in run r_3 (since p_i does not UR-Deliver m at all). However, in run r_3, p_i knows that it UR-Delivered some message m, therefore at least two forced logs are mandatory to UR-Deliver a message m. □

Proposition 19. *In a system with n_c correct processes, if each correct process p_i UR-Delivers a message, then each p_i performs at least one forced log and at least n_c+1 forced logs have been performed in the system.*

Proof. Follows directly from lemmata 17 and 18. □

Proposition 20. *Consider any strongly uniform reliable broadcast algorithm A; let p_i be any*

process, if p_i SUR-Delivers a message m, then at least n-n_c+2 forced logs have been performed in the system (where n-n_c is the number of faulty processes in the system).

Proof. With lemma 18, if a correct process p_i UR-Delivers a message m, then two forced logs have been performed in the system; this is also clearly the case when p_i SUR-Delivers m. However, it is not sufficient, consider the following case. Process p_i SUTO-Broadcasts m, SUTO-Delivers m, all processes crash and p_i never recovers. To satisfy property A.3, every correct process should SUTO-Deliver m. However, this is impossible since no process that is up has kept track of m, therefore more than two forced logs in the system are required. In fact, n-n_c+1 forced logs are mandatory (one per process), since we want to ensure that at least one correct process keeps track of m in case all processes crash. From lemma 17, we know that at least one forced log is mandatory before returning from UTO-Broadcast, it is also trivially the case for the primitive SUTO-Broadcast. The number of required forced logs performed in the system when a process SUTO-Delivers a message is therefore at least, n-n_c+1+1 = n-n_c+2. □

5.2 Total Order Broadcast

First, we describe in Figure 14 a simple adaptation to fair lossy channels of the total order broadcast algorithm of [7]. The adapted algorithm satisfies the weakest of our total order broadcast specification (V.1, A.1, I.1, TO.1). The algorithm uses a series of consecutive consensus instances: each consensus instance being used to agree on a batch of messages. Each process differentiates consecutive instances by maintaining a local counter (k): each value of the counter corresponds to a specific consensus instance. We describe first the main data structure of the algorithm. A local set *Received* keeps track of all messages that needs to be decided, and another set *TO_Delivered* keeps track of all TO-Delivered messages. Intuitively, the algorithm works as follows for a given process p_i. When there are still messages to be TO-Delivered, i.e., *Received*-*TO_Delivered* is not empty, process p_i launches a consensus instance and waits for the decision value of consensus. Once p_i s-receives the decision, p_i removes all TO-Delivered messages from the batch and atomically deliver all the messages. Note that, once p_i TO-Delivers m, then p_i R-Broadcasts the delivered messages to every process in order to satisfy property A.1 of total order broadcast. We assume for the rest of the section that there is a majority of correct processes in the system.[9] Due to a lack of space, we give the correctness proofs of the algorithm of Figure 14 in optional Appendix A.

[9]Due to a lack of space, we omit to show that the consensus implementation of [7] adapted to fair lossy channels, i.e., use of retransmission module and added recovery procedure, satisfies the properties of consensus in

```
1: For each process p_i:
2: procedure initialisation:
3:    Received[] ← ⊥; k ← 0; TO_Delivered[] ← ⊥
4: upon TO-Broadcast(m) do
5:    R-Broadcast(m)
6: upon R-Deliver(m) do
7:    Received ← Received ∪ m
8: TO-Deliver(k) occurs as follows:
9:    while Received - TO_Delivered ≠ ⊥ do
10:      k ← k + 1; propose(k, Received- TO_Delivered)
11:      wait until[receive(decide(k, msgSet^k))]
12:      TO_Delivered^k ← msgSet^k - TO_Delivered
13:      atomically deliver all messages in TO_Delivered^k in some deterministic order      {TO-Deliver m}
14:      TO_Delivered ← TO_Delivered ∪ TO_Delivered^k
15:      R-Broadcast(msgSet^k)                                                              {Added from [7]}
16: upon recovery do
17:    initialisation
```

Figure 14: Adaptation of the total order broadcast of [7]

Uniform Total Order Broadcast. For total order broadcast algorithms, the optimisation pattern used for reliable broadcast is not sufficient and cannot be applied. Consider the following case depicted in Figure 15. Process p_1 UTO-Broadcasts m but only process p_2 UTO-Delivers m; p_2 then UTO-Broadcasts m' and UTO-Delivers m', therefore all processes should UTO-Deliver m and then m'. If p_2 simply s-sends m', p_1 can first UTO-Deliver m' and then m, which violates property TO.2. This example explains why the retransmission module is not sufficient to implement total order broadcast: an agreement phase is mandatory.

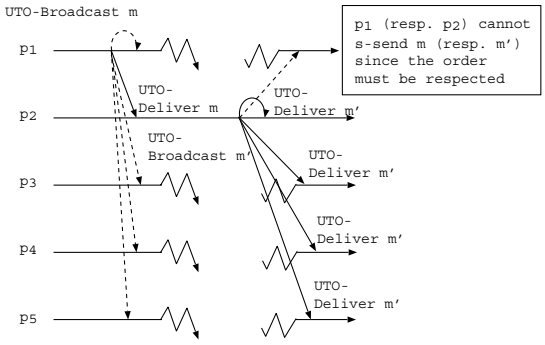


Figure 15: Retransmission module is not sufficient

We give here the intuition underlying the total order algorithms obtained after our transformations and optimisations. As shown by the example of Figure 15, the optimisation pattern for reliable broadcast is different from the optimisation pattern for total order broadcast such that the following steps are imposed:

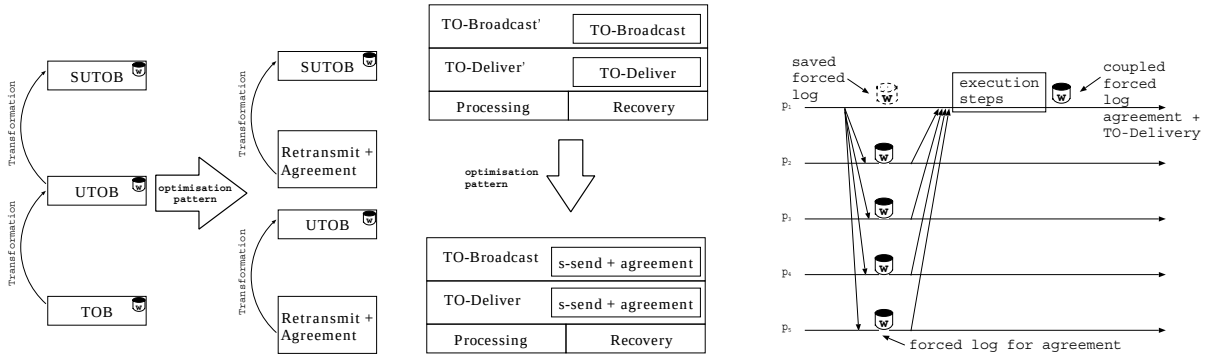
- An agreement phase is added to the retransmission module that replaces the middle layer

our system model. Moreover, we assume here the reliable broadcast algorithm of [15] of Figure 9.

(A).

- The process that coordinates the agreement phase saves one forced log by coupling the forced log of the agreement together with the forced log of the TO-Delivery.

Figure 16 presents the optimisation pattern for total order broadcast. As described in Figure 16(c), the agreement phase can be improved since performing one forced log for the agreement and one forced log for the TO-Delivery for every process is not mandatory. Instead, the coordinator process waits for $\lceil \frac{n}{2} \rceil$ process replies (other than itself), executes some steps, and then performs one forced log that couples the forced log required for the agreement and the forced log of the TO-Delivery. Every other process executes the usual scheme, one forced log for the agreement and one forced log for the TO-Delivery.



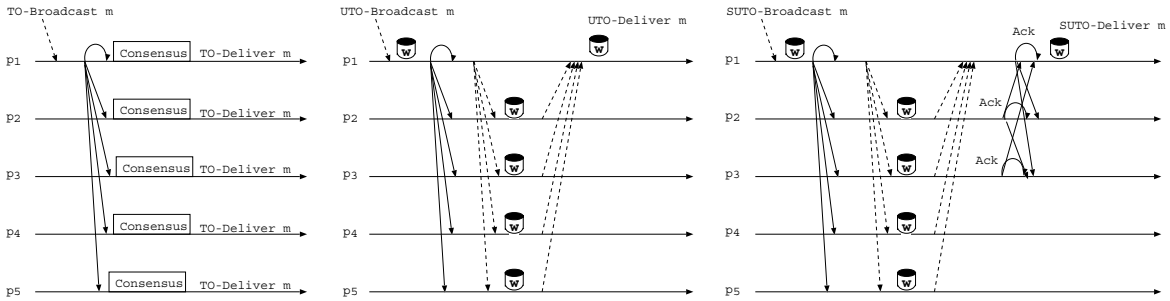
(a) Merging layers (b) Replacing primitives (c) Improved agreement phase

Figure 16: Optimisation pattern for total order broadcast

Strongly Uniform Total Order Broadcast. Our strongly uniform total order broadcast algorithm is the result of the total order optimisation pattern applied to transformer $T_{UTO \rightarrow SUTOB}$. The only difference with the uniform total order algorithm is in the way a process p_i TO-Delivers messages, since p_i needs to wait for a majority of processes to acknowledge a batch before TO-Delivering it.

Analytical Performance and Lower Bounds. Figure 17 depicts the communication and stable storage pattern, in nice runs, of several total order algorithms: (a) the total order broadcast of [7], (b) the uniform total order broadcast and (c) the strongly uniform total order broadcast. For presentation clarity, for both uniform broadcasts, Figures 17(b) and 17(c) depict only the agreement phase of process p_1 (in dots), and, in addition, Figure 17(c) depicts the necessary acknowledgements for p_1 to SUTO-Deliver a message. Figure 17(a) shows that the total order broadcast algorithm does not perform any forced log. Figure 17 depicts, for uniform total order

broadcast, that a process p_i performs one forced log when UTO-Broadcasting a message m, and one forced log when UTO-Delivering m. Moreover, every process (except the coordinator of the agreement phase) performs an additional forced log for the agreement part. Strongly uniform total order broadcast has the same forced log pattern than uniform total order broadcast but performs the forced log of the TO-Delivery once a majority of processes has acknowledged the message. We state for uniform total order broadcast that, if a process p_i UTO-Delivers a batch of messages, then (i) $\lceil \frac{n}{2} \rceil$ processes (including p_i) have performed one forced log, and (ii) $\lceil \frac{n}{2} \rceil$+1 forced logs have been performed in the system.



(a) Regular (b) Uniform (c) Strongly uniform

Figure 17: Communication and stable storage pattern for total order broadcast

Lemma 21. *Consider any uniform total order broadcast algorithm A; let p_i be any process, If p_i UTO-Delivers a batch of messages then at least $\lceil \frac{n}{2} \rceil$ (including p_i) processes have performed one forced log and at least $\lceil \frac{n}{2} \rceil$+1 forced logs have been performed in the system.*

Proof. As shown in lemma 17, the process that UTO-Broadcasts must perform one forced log before returning from UTO-Broadcast. Assume that A solves uniform total order broadcast with $\lceil \frac{n}{2} \rceil$-1 forced logs (one forced log per process). Suppose now that only the faulty ($\lceil \frac{n}{2} \rceil$-1) processes UTO-Deliver batch k, i.e., store batch k, crash and never recover. A correct process p_i can then decide another value for batch k, i.e., the property TO.2 of A is violated. Therefore, if $\lceil \frac{n}{2} \rceil$ forced logs are performed (one forced log per process), at least one correct process has stored batch k since there at most $\frac{n}{2}$ faulty processes in the system. Indeed, at least $\lceil \frac{n}{2} \rceil$+1 forced logs have been performed in the system. □

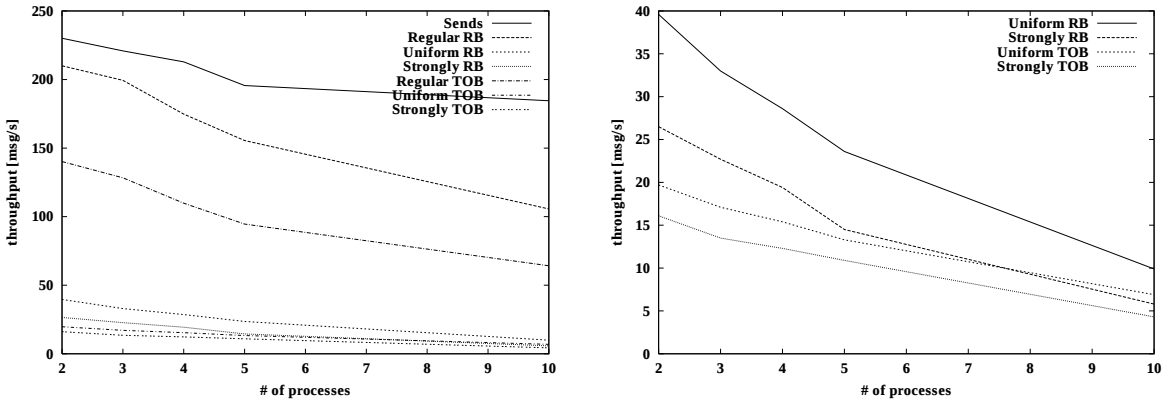
Proposition 22. *In a system with n_c correct processes, if each correct process p_i UTO-Delivers a batch of messages, then at least $2n_c$ forced logs have been performed in the system.*

Proof. Follows directly from lemma 21 and the fact that (a) one forced log is mandatory before returning from the primitive UTO-Broadcast, and (b) one forced log is mandatory for

the coordinator of the agreement phase to UTO-Deliver a message. The minimal number of forced logs is performed if both previous forced logs are performed on the same correct process. Then every other correct process (n_c-1) performs two forced logs to UTO-Deliver m, we have then $2(n_c\text{-}1)+1+1 = 2n_c$. □

5.3 Experimental Measures

We give some practical performance measurements of the algorithms that result from our transformations and optimisations. Our measurements reflect the impact of uniformity on the actual performance. These measurements were made on a LAN interconnected by Fast Ethernet (10MB/s) on a normal working day. The LAN consisted of 60 UltraSUN 10 (256Mb RAM, 9 Gb Harddisk) machines. All stations were running Solaris 2.7, and our implementation was running Solaris JVM (JDK 1.2.1, native threads, JIT). The effective message size was of 1Kb and the performance tests consider only cases where as many broadcasts as possible are executed. When the number of processes increases, not only the number of recipients increases but also the number of broadcasting processes. These tests consider nice runs: no process or link crashes or is suspected to have crashed; these runs are the most frequent in practice, and are those for which algorithms are usually optimised. Figure 18(a) summarises the results of the throughput measurements for each type of broadcast. Not surprisingly, our comparison depicts the fact that the more forced logs a broadcast contains (stronger specification), the worst the throughput is. We give a more detailed view of the results in Figure 18(b).



(a) All broadcasts

(b) Uniform and strongly uniform broadcasts

Figure 18: Performance comparison

To measure the overhead of uniformity, we have performed simple message sends between

processes (until we reach the network capacity). This performance test measures the overhead of the retransmission module. We can also figure out that the existing reliable (resp. total order) broadcasts for the crash-stop model should have a performance that lies between the two top (resp. top and third) curves of Figure 18(a) since we assume that known crash-stop implementations must be as efficient as our implementation. As conveyed by the measurement results, the performance of the reliable and total order broadcasts are by far better than the ones requiring stable storage (i.e., uniformity). Figure 18(b) is a bit misleading since it gives the impression that, for ten processes, the performance varies very little for broadcasts requiring forced logs. In fact, the scale is really large and the difference is quite noticeable; the measures are given in Figure 19. Note that for both type of broadcast (reliable and total order), the uniform and strongly uniform versions are limited by the overhead time that it takes to store messages on stable storage. On our workstations, a forced log of the size of 1Kb took in average around 60 milliseconds.[10] On the other hand, the performance of the weakest broadcasts (without forced log) are limited by the overall performance of the network, which is conveyed by the quickly decreasing throughput. Again, due to the overhead of the stable storage, we notice that (i) reliable and total order broadcasts of the same type have performance close to each other, and (ii) the communication overhead is almost negligible. These results confirm that forced logs are a major overhead compared to communication steps and should be avoided as much as possible.

A solution to reduce the number of forced logs is to (reliable or total order) broadcast using batch of messages, i.e., only one forced log is performed for numerous messages. Note that it takes about half a second to perform a forced log of the size of 100 Kb which is equivalent to a rate of around 200 msg/sec. When fine-tuning our total order broadcast algorithms, we also found out that starting too many concurrent instances of consensus had a dramatic impact on the throughput. The best performance presented here are obtained with consecutive consensus instances at a rate of 1 or 2 consensus instance per second.

6 Concluding Remarks

We position here our specifications and algorithms with respect to related work.

Pragmatic approaches. Considerable work has been devoted to the implementation of broadcast primitives in practical system models where processes and channels may crash and recover,

[10]Note that it is a synchronous forced log, e.g., in Java this requires more than just writing on a disk since this operation only writes data to the cache memory. Instead, we had to force the log by using C code that directly accesses the disk.

Processes		Reliable			Total Order		
	Sends	Regular	Uniform	Strongly	Regular	Uniform	Strongly
2	230	210	40	27	140	20	16
3	221	199	33	23	128	17	14
4	213	175	29	20	109	15	12
5	196	155	24	15	95	13	11
10	184	106	10	6	64	7	4

Figure 19: Throughput figures (msg/sec)

e.g., MTP [2], RMP [23], RBP [8], TRAM [9], RMTP [18], and TMTP [24]. These broadcast algorithms do not aim at ensuring agreement in all possible situations [4]: if the sender of a message crashes, some processes might deliver the message whereas others might not. In fact, agreement is ensured on a best effort basis. The motivation of our work was precisely to figure out what it takes to always ensure agreement and total order in a practical crash-recovery system model.

Group communication systems like Isis [5, 21], Transis [11], or Totem [20] indirectly address the crash-recovery issue through a *group membership* abstraction. A process that crashes is excluded from the group and, when it recovers, it rejoins the group. Message delivery is synchronised with view changes through the notion of *view synchrony* and, roughly speaking, a process that leaves the group is exempted from delivering a message. In some sense, the guarantees offered by these systems are weaker than those corresponding to our specifications. For instance, we require that any correct process (even if it crashes and recovers) eventually delivers every message delivered by a correct process. On the other hand, view synchrony provides a notion of process exemption and a process that crashes is excluded from the group; hence, this process is not required to deliver every message.[11] One can circumvent the issue by assuming that a process that recovers changes its identity, but the problem is then postponed to the application level.

In short, many practitioners considered the problem of broadcasting messages in a crash-recovery system model. The algorithms proposed obviously ensure weaker guarantees than the specifications of our primitives, and finding out the actual specifications of the primitives implemented by those algorithms is an interesting issue. A complementary interesting question is how to devise crash-recovery broadcast algorithms that satisfy the specifications we defined in this paper, in a probabilistic manner, e.g., along the lines of [4, 12].

[11] In practice, if the process recovers and rejoins the group, a state transfer mechanism is performed to update the state of the newcomer. However, it is not clear to capture the actual guarantee offered by the state transfer mechanism.

Traditional specifications. The only comprehensive study of fault-tolerant broadcast specifications we are aware of is [15]. In [15], the authors consider different kinds of process failures: roughly speaking, *crash* failures model the definite halting of activities, *omission* failures model the skipping of messages, and *Byzantine* failures model arbitrary behaviour.

- One might draw an interesting analogy between the omission failure model and the crash-recovery model. Indeed, just like in an omission failure model, a process p_i can be up at a given time, yet p_i could have lost all messages sent to it before t: just like if p_i had crashed and recovered by t. However, in a crash-recovery model, p_i would have lost not only the messages it received, but also its volatile state. Hence, the analogy would be accurate if we assume that processes store every state change to stable storage (i.e., exclusively use stable storage vs volatile memory), but this would be a very expensive analogy. One of our objectives when designing algorithms in a crash-recovery model is precisely to minimise the access to stable storage. Hence, omissions capture only one aspect of the actual failure that a process might commit in a crash-recovery model.

- The Byzantine failure model could be viewed as more general that the crash-recovery model and one would wonder whether the specifications and algorithms devised in the Byzantine model could be used in a crash-recovery model. Indeed, a process that crashes and recovers can obviously be viewed as a Byzantine process. However, in our crash-recovery resilient broadcast specifications, processes that crash and recover several times, yet that eventually remain up, are considered correct and are supposed to behave in a consistent manner, e.g., they are required to deliver messages that have been broadcast by correct processes. One cannot make any such requirement on any arbitrary Byzantine process. As a consequence, specifications of Byzantine resilient broadcast primitives simply do not fit a crash-recovery model.

Crash-recovery resilient total order broadcasts. We know of two broadcast algorithms that ensure strong reliability in a practical crash-recovery system model: the algorithm of [22] and the algorithm of [17]. Both ensure total order delivery of messages. In fact, these two algorithms ensure our strongly uniform total order broadcast specification (i.e., properties V.2, A.3, I.3 and TO.3). The algorithm of [22] is modular in the sense that it relies on an underlying consensus abstraction. The algorithm of [17] opens that abstraction for performance reasons. In fact, the strongest of our total order algorithms that we obtain from our transformations and optimisations corresponds exactly to the algorithm of [17].

References

[1] M. K. Aguilera, W. Chen, and S. Toueg. Failure detection and consensus in the crash-recovery model. *Distributed Computing*, 13(2):99–125, May 2000.

[2] S. Armstrong, A. Freier, and K. Marzullo. Multicast transport protocol. Internet Request for Comments RFC 1301, February 1992.

[3] P. A Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[4] K. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu, and Y. Minsky. Bimodal multicast. *ACM Transactions on Computer Systems*, 17(2):41–88, May 1999.

[5] K. Birman and R. van Renessee. *Reliable Distributed Computing with the Isis Toolkit*. IEEE Computer Society Press, 1993.

[6] R. Boichat and R. Guerraoui. Reliable broadcast in the crash-recovery model. In *Proceedings of the 19th IEEE Symposium on Reliable Distributed Systems (SRDS-19)*, pages 98–107, Nuerenberg, Germany, October 2000.

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

[8] J. Chang and N. F. Maxemchuck. Reliable broadcast protocols. *ACM Transactions on Computer Systems*, 2(3):251–273, August 1984.

[9] D.-M. Chiu, S. Hurst, M. Kadansky, and J. Wesley. Tram : A tree-based reliable multicast protocol. Technical Report TR-98-66, Sun Microsystems, July 1998.

[10] D. Dolev, S. Kramer, and D. Malkhi. Early delivery totally ordered broadcast in asynchronous environments. In *Proceedings of the 23rd IEEE International Symposium on Fault-Tolerant Computing (FTCS-23)*, pages 296–306, Toulouse, France, June 1993.

[11] D. Dolev and D. Malkhi. The transis approach to high-availability cluster communication. *Communications of the ACM*, 39(4):64–70, April 1996.

[12] P.Th. Eugster, R. Guerraoui, S. Handurukande, A.-M. Kermarrec, and P. Kouznetsov. Lightweight probabilistic broadcast. In *Proceedings of the 2nd IEEE International Conference on Dependable Systems and Networks (DSN-2)*, Göteborg, Sweden, July 2001.

[13] P. Ezhilchelvan, R. Macedo, and S. Shrivastava. Newtop: A fault-tolerant group communication protocol. In *Proceedings of the 15th IEEE International Conference on Distributed Computing Systems (ICDCS-15)*, pages 296–306, Vancouver, Canada, May 1995.

[14] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[15] V. Hadzilacos and S. Toueg. Fault-tolerant broadcasts and related problems. In S. Mullender, editor, *Distributed Systems*, ACM Press Books, chapter 5, pages 97–146. Addison-Wesley, second edition, 1993.

[16] M. Hiltunen and R. Schlichting. An approach to constructing modular fault-tolerant protocols. In *Proceedings of the 12th IEEE Symposium on Reliable Distributed Systems (SRDS-12)*, pages 105–114, Princeton, NJ, USA, 1993.

[17] L. Lamport. The part-time parliament. Technical Report 49, Systems Research Center, Digital Equipement Corp, Palo Alto, September 1989. A revised version of the paper also appeared in ACM Transaction on Computer Systems vol.16 number 2.

[18] J.C. Lin and S. Paul. Rmtp: A reliable multicast transport protocol. In *Proceedings of the 15th IEEE Conference on Computer Communications (INFOCOM-15)*, pages 1414–1424, March 1996.

[19] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[20] L.E. Moser, P.M. Meillar-Smith, D.A. Agarwal, R.K. Budhia, and C.A. Lingley-Papadopoulos. Totem : A fault-tolerant multicast group communication system. *Communications of the ACM*, 39(4):54–63, April 1996.

[21] D. Powell. Group communications. *Communications of the ACM*, 39(4):50–97, April 1996.

[22] L. Rodrigues and M. Raynal. Atomic broadcast in asynchronous systems where processes can crash and recover. In *Proceedings of the 20th IEEE International Conference on Distributed Computing Systems (ICDCS-20)*, pages 288–295, Taipei, Taiwan, April 2000.

[23] B. Whetten, T. Montgomery, and S. Kaplan. A high performance totally ordered multicast protocol. In *Theory and Practice in Distributed Systems*, volume 938 of *Lecture Notes in Computer Science*, pages 33–57. Springer-Verlag, 1994.

[24] R. Yavatkar, J.Griffoen, and M. Sudan. A reliable dissemination protocol for interactive collaborative applications. In *Proceedings of the 3rd ACM ACM International Conference on Multimedia*, pages 333–344, San Francisco, CA, USA, November 1995.

A Optional Appendix: Total Order Broadcast Proofs

This appendix presents the correctness proofs for transformers $T_{TOB \rightarrow UTOB}$ and $T_{UTOB \rightarrow SUTOB}$, and for the total order broadcast of [7] adapted to fair-lossy channels.

A.1 Transformer $T_{TOB \rightarrow UTOB}$

Lemma 23. *The algorithm of Figure 7 transforms the property V.1 of A into the property V.2 of A′: If a correct process p_i TO'-Broadcasts a message m, then p_i eventually TO'-Delivers m.*

Proof. Suppose that a correct process p_i TO'-Broadcasts m and assume by contradiction that p_i never TO'-Delivers m (i.e., p_i violates property V.2). There are two cases to consider: (i) p_i does not crash, and (ii) p_i crashes, recovers and remains always-up. For case (i), since p_i does not crash, then by the property V.1 of A, p_i eventually TO-Delivers m (line 21). Process p_i then either TO'-Delivers m at line 10 if it is the next batch that p_i is waiting for, or TO'-Delivers m later in a next batch of messages at line 14: a contradiction. For case (ii), there is a time after which p_i stops crashing and remains always-up. When p_i recovers, p_i retrieves either (a) the batches of messages that it TO-Delivered but never TO'-Delivered, or (b) the batches of messages that it TO'-Broadcasts at line 26 (*msgSent*). For case (a), p_i then TO'-Delivers these batches of messages (one containing m) if they are the next batches of messages that p_i is waiting for, or TO'-Delivers them later: a contradiction. For case (b), p_i then TO-Broadcasts these batches at line 28. We are certain that p_i has stored m into stable storage at line 19 since a process has TO'-Broadcast m only when it returns from the invocation of TO'-Broadcast. By line 19 of the algorithm, we ensure that the forced log will be executed before returning from the invocation of TO'-Broadcast. Since p_i has never TO-Delivered m, then by the property V.1 of A, p_i TO-Delivers m at line 21 and then TO'-Delivers m if it is part of the next batch of messages that p_i is waiting for, or TO'-Delivers m later: a contradiction. □

Lemma 24. *The algorithm of Figure 7 transforms the property A.1 of A into the property A.2 of A′: If a correct process TO'-Delivers a message m, then every correct process eventually TO'-Delivers m.*

Proof. Suppose that p_i is a correct process that TO'-Delivers m and assume by contradiction that p_j is a correct process that does not TO'-Deliver m. There are four cases to consider: (a) p_i and p_j do not crash, or (b) p_i crashes, recovers and remains always-up, and p_j does not crash, or (c) p_i does not crash, and p_j crashes, recovers and remains always-up, and finally (d) both

processes p_i and p_j crash, recover and remain eventually always-up. For case (a), since both processes do not crash, by the property A.1 of A, p_j TO-Delivers m, therefore p_j TO'-Delivers m if it is part of the next batch of messages that p_i is waiting for, or TO'-Delivers m later: a contradiction. For case (b), since p_i is correct, there is a time after p_i stops crashing and remains always-up. After recovering, p_i retrieves the messages that it TO'-Delivered at line 26, and s-sends them to every process at lines 31-32. By the validity property of the retransmission module, p_j eventually s-receives the TO'-Delivered from p_i at line 23, and then TO'-Delivers m if it is part of the next batch of messages that p_i is waiting for, or TO'-Delivers m later: a contradiction. For case (c), since p_j is correct, there is a time after which p_j stops crashing and remains always-up. Before TO'-Delivering m, p_i s-sends the batch of messages containing m to p_j at line 22. As for case (b), p_j then eventually s-receives m, and then TO'-Delivers m if it is part of the next batch of messages that p_i is waiting for, or TO'-Delivers m later: a contradiction. Finally, case (d) is a mix of case (b) and (c); there is a time after which both processes p_i and p_j stop crashing and remain always-up. As for case (b) and (c), p_i s-sends a batch of messages containing m to p_j, p_j then eventually s-receives it and then TO'-Delivers m if it is part of the next batch of messages that p_i is waiting for, or TO'-Delivers m later: a contradiction. □

Lemma 25. *The algorithm of Figure 7 transforms the property I.1 of A into the property I.2 of A′: For any message m, every correct process p_i TO'-Delivers m at most once, and only if m was previously TO'-Broadcast by sender(m).*

Proof. For the first part of the property, suppose by contradiction that a correct process p_i TO'-Delivers m more than once. We have two cases to consider: (i) p_i does not crash, or (ii) p_i crashes, recovers and remains always-up. Case (i) is clearly impossible since before TO'-Delivering m, p_i appends the batch of messages (containing m) to the set *to'_delivered* at line 8, and checks for the batch of messages in the set *to'_delivered* at guard line 7: a contradiction. For case (ii), there is a time after which p_i stops crashing and remains always-up. Remember that p_i stores the set *to'_delivered* and the variable *nextBatch* into stable storage at line 10. When p_i crashes and recovers, p_i retrieves the set *to'_delivered* and the variable *nextBatch*. Therefore, p_i cannot go through guard line 7 twice since p_i checks that (a) the set *to'_delivered*[*nextBatch*] is empty, and (b) the k^{th} batch of messages that is TO'-Delivered is equal to *nextBatch*: a contradiction. The second part follows from the no creation property of the channels. This property prevents the case of s-receiving (resp. TO-Delivering) messages that were not s-sent (resp. TO-Broadcast). □

Lemma 26. *The algorithm of Figure 7 transforms the property TO.1 of A into the property TO.2 of A′: If correct processes p_i and p_j both TO'-Deliver m and m', then p_i TO'-Delivers m before m' if and only if p_j TO'-Delivers m before m'.*

Proof. Suppose that a correct process p_i TO'-Delivers m before m' and assume by contradiction that another correct process p_j TO'-Delivers m' before m. Since we assume that there is a deterministic rule for TO'-Delivering a batch of messages, we know that if this situation occurs, m and m' have to be in two different batches of messages. Assume that m is in the k^{th} batch of messages and m' in the $k{+}1^{th}$ batch of messages. There are now two cases to consider: (i) p_i and p_j do not crash, or (ii) one of them (p_i and p_j) crashes, recovers and remains always-up. For case (i), by the algorithm of Figure 7, if p_i TO'-Delivers m before m', then p_j cannot TO'-Deliver m' before m since guard line 7 forbids p_j to TO'-Deliver batches of messages out of order. Process p_j could s-receive the $k{+}1^{th}$ batch of messages before TO-Delivering the k^{th} batch of messages, however guard line 7 prevents p_j from TO'-Delivering the $k{+}1^{th}$ batch of messages before the k^{th} batch of messages: a contradiction. For case (ii), there is a time after which p_i or p_j stops crashing and remains always-up. Remember that a process stores the k^{th} batch of messages into stable storage (line 10) before TO'-Delivering it. When a process recovers, it retrieves and s-sends the messages that it TO'-Delivered along with the value of the next expected batch of messages. This precludes a correct process to wait for a batch of messages that it already TO'-Delivered; and for the same reasons described for case (i), a process cannot TO'-Deliver batches out of order: a contradiction. □

Proposition 27. *The algorithm of Figure 7 transforms a total order broadcast into a uniform total order broadcast.*

Proof. Follows directly from lemmata 23, 24, 25 and 26. □

A.2 Transformer $T_{UTOB \rightarrow SUTOB}$

Lemma 28. *The algorithm of Figure 8 preserves the property V.2 of A into A′: If a correct process p_i TO'-Broadcasts a message m, then p_i eventually TO'-Delivers m.*

Proof. Suppose by contradiction that a correct process p_i TO'-Broadcasts m and never TO'-Delivers m (i.e., p_i violates property V.2). There are two cases to consider: (i) p_i does not crash, or (ii) p_i crashes, recovers and remains always-up. For case (i), since (a) p_i TO-Broadcasts m and waits for a majority of ACK(k) where k is the batch of messages containing m, (b) there is a majority of correct processes in the system, (c) there is a time after which those correct

processes stop crashing and remain always-up, and (d) by the property A.2 of A, eventually every correct process TO-Delivers m, and then acknowledges it by s-sending an ACK(k). By the validity property of the retransmission module, p_i then s-receives a majority of ACK(k) and TO'-Delivers m if it is part of the next batch of messages that p_i is waiting for, or TO'-Delivers m later: a contradiction. For case (ii), there is a time after which p_i stops crashing and remains always-up, then (a) p_i retrieves and s-sends, when recovering, the messages it TO-Delivers but did not TO'-Deliver, and (b) for the same reasons invoked for case (i), p_i eventually s-receives a majority of ACK(k) and therefore TO'-Delivers this batch of messages if it is the next batch of messages that p_i is waiting for, or TO'-Delivers it later: a contradiction. □

Lemma 29. *The algorithm of Figure 8 transforms the property A.2 of A into the property A.3 of A': If a process TO'-Delivers a message m, then every correct process eventually TO'-Delivers m.*

Proof. Let p_i be a process that TO'-Delivers a batch of messages containing m and assume by contradiction that a correct process p_j does not TO'-Deliver m. When p_i TO'-Delivers m, p_i has to s-receive a majority of ACK(k) at line 23. Since we assume a majority of correct processes, there is at least one correct process p_j that has TO-Delivered the batch of messages k that contains m since ACK(k) messages are only s-sent when a process has TO-Delivered batch k (line 19). There is a time after which every correct process stops crashing and remains always-up. Since p_j TO-Delivered batch k, by the property A.2 of A, every correct process TO-Delivers batch k and then acknowledges k when recovering. Hence, p_j s-receives a majority of ACK(k) and TO'-Delivers k which includes m: a contradiction. □

Lemma 30. *The algorithm of Figure 8 preserves the property I.2 of A into A': For any message m, every correct process p_i TO'-Delivers m at most once, and only if m was previously TO'-Broadcast by sender(m).*

Proof. Identical to the proof of lemma 25. □

Lemma 31. *The algorithm in Figure 8 preserves the property TO.2 of A into A': If correct processes p_i and p_j both TO'-Deliver m and m', then p_i TO'-Delivers m before m' if and only if p_j TO'-Delivers m before m'.*

Proof. Identical to the proof of lemma 26. □

Proposition 32. *The algorithm of Figure 5 transforms a uniform total order broadcast into a strongly uniform total order broadcast.*

Proof. Follows directly from lemmata 29, 30, 31 and 32. □

A.3 Total Order Algorithm

Lemma 33. *The algorithm of Figure 14 satisfies the property V.1 of total order broadcast: If a process p_i TO-Broadcasts a message m and then does not crash, p_i eventually TO-Delivers m.*

Proof. Suppose by contradiction that p_i TO-Broadcasts a message m, then does not crash and never TO-Delivers m. Since we assume a majority of correct processes, there is a time after which a majority of correct processes stops from crashing and remains always-up. By the algorithm of Figure 14, since once p_i TO-Broadcasts m, p_i does not crash, then p_i R-Broadcasts m, R-Delivers m and then proposes m in a batch. Eventually, p_i receives the decision and then TO-Delivers m: a contradiction. □

Lemma 34. *The algorithm of Figure 14 satisfies the property A.1 of total order broadcast: If a process p_i TO-Delivers a message m and then does not crash, then any process that does not crash after p_i TO-Delivers m eventually TO-Delivers m.*

Proof. Let p_i be a process p_i that TO-Delivers m and does not crash afterwards, let p_j be a process that does not crash after p_i TO-Delivers m and assume by contradiction that p_j never TO-Delivers m. When p_i TO-Delivers m, there is a modified step compared to [7] since p_i R-Broadcasts again m to ensure that all correct processes that do not crash after p_i TO-Delivers m, eventually TO-Deliver m. By the property A.1 of reliable broadcast, all processes that do not crash after p_i TO-Deliver m, R-Deliver m, then eventually propose m, decide m and TO-Deliver m: a contradiction. □

Lemma 35. *The algorithm of Figure 14 satisfies the property I.1 of total order broadcast: For any message m, every process p_i that TO-Delivers m and then does not crash, TO-Delivers m at most once, and only if m was previously TO-Broadcast by sender(m).*

Proof. Assume by contradiction that a process p_i TO-Delivers a message m and then does not crash. By the no creation property of the channels, p_i cannot TO-Deliver a message out of thin air, and since p_i does not crash after TO-Delivering m, line 12 prevents p_i from TO-Delivering m twice: a contradiction. □

Lemma 36. *The algorithm of Figure 14 satisfies the property TO.1 of total order broadcast: Let p_i and p_j be any two processes that TO-Deliver some message m. If p_i TO-Delivers some message m' before m and then does not crash, then if p_j also TO-Delivers m' and then does not*

crash, p_j TO-Delivers m' before m.

Proof. Assume that processes p_i and p_j TO-Deliver a message m' and then do not crash. Assume by contradiction that p_i TO-Delivers m after m' and p_j never TO-Delivers m. When p_i TO-Delivers m, p_i R-Broadcasts m, then by the property A.1 of reliable broadcast and the algorithm of Figure 14, p_j TO-Delivers m after m': a contradiction. □

Proposition 37: *The algorithm of Figure 14 satisfies the properties V.1, A.1, I.1 and TO.1 of total order broadcast.*

Proof. Follows directly from lemmata 33, 34, 35 and 36. □