Dynamic Byzantine Reliable Broadcast

Rachid Guerraoui
EPFL, Lausanne, Switzerland

Jovan Komatovic
EPFL, Lausanne, Switzerland

Petr Kuznetsov
LTCI, Télécom Paris
Institut Polytechnique Paris, France

Yvonne-Anne Pignolet
DFINITY Foundation, Zürich, Switzerland

Dragos-Adrian Seredinschi
Informal Systems, Lausanne, Switzerland

Andrei Tonkikh
National Research University Higher School of Economics, St. Petersburg, Russia

Abstract

Reliable broadcast is a communication primitive guaranteeing, intuitively, that all processes in a distributed system deliver the same set of messages. The reason why this primitive is appealing is twofold: *(i)* we can implement it deterministically in a completely asynchronous environment, unlike stronger primitives like consensus and total-order broadcast, and yet *(ii)* reliable broadcast is powerful enough to implement important applications like payment systems.

The problem we tackle in this paper is that of *dynamic* reliable broadcast, i.e., enabling processes to join or leave the system. This property is desirable for long-lived applications (aiming to be highly available), yet has been precluded in previous asynchronous reliable broadcast protocols. We study this property in a general adversarial (i.e., Byzantine) environment.

We introduce the first specification of a dynamic Byzantine reliable broadcast (DBRB) primitive that is amenable to an asynchronous implementation. We then present an algorithm implementing this specification in an asynchronous network. Our DBRB algorithm ensures that if any correct process in the system broadcasts a message, then every correct process delivers that message unless it leaves the system. Moreover, if a correct process delivers a message, then every correct process that has not expressed its will to leave the system delivers that message. We assume that more than 2/3 of processes in the system are correct at all times, which is tight in our context.

We also show that if only one process in the system can fail – and it can fail only by crashing – then it is impossible to implement a stronger primitive, ensuring that if any correct process in the system broadcasts or delivers a message, then every correct process in the system delivers that message – including those that leave.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Byzantine reliable broadcast, deterministic distributed algorithms, dynamic distributed systems

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2020.23

Related Version A full version of the paper is available at `https://arxiv.org/abs/2001.06271`.

Funding This Work Has Been Supported in Part by the Interchain Foundation, Cross-Chain Validation Project.

1 Introduction

Networks typically offer a reliable form of communication channels: TCP. As an abstraction, these channels ensure that if neither the sender nor the destination of a message fail, then the message is eventually delivered. Essentially, this abstraction hides the unreliability of the underlying IP layer, so the user of a TCP channel is unaware of the lost messages.

Yet, for many applications, TCP is not reliable enough. Indeed, think of the situation where a message needs to be sent to all processes of a distributed system. If the sender does not fail, TCP will do the job; but otherwise, the message might reach only a strict subset of processes. This can be problematic for certain applications, such as a financial notification service when processes subscribe to information published by other processes. For fairness reasons, one might want to ensure that if the sender fails, either *all or no process* delivers that message. Moreover, if the correct processes choose to deliver, they must deliver the same message, even when the sender is Byzantine. We talk, therefore, about *reliable broadcast*. Such a primitive does not ensure that messages are delivered in the same total order, but simply in the “all-or-nothing” manner.

Reliable broadcast is handy for many applications, including, for example, cryptocurrencies. Indeed, in contrast to what was implicitly considered since Nakamoto’s original paper [24], there is no need to ensure consensus on the ordering of messages, i.e., to totally order messages, if the goal is to perform secure payments. A reliable broadcast scheme suffices [15].

Reliable broadcast is also attractive because, unlike stronger primitives such as total order broadcast and consensus, it can be implemented deterministically in a completely asynchronous environment [7]. The basic idea uses a quorum of correct processes, and makes that quorum responsible for ensuring that a message is transmitted to all processes if the original sender of the message fails. If a message does not reach the quorum, it will not be delivered by any process. It is important to notice at this point a terminology difference between the act of “receiving” and the act of “delivering” a message. A process indeed might “receive” a message m, but not necessarily “deliver” m to its application until it is confident that the “all-or-nothing” property of the reliable broadcast is ensured.

A closer look at prior asynchronous implementations of reliable broadcast reveals, however, a gap between theory and practice. The implementations described so far all assume a *static* system. Essentially, the set of processes in the system remains the same, except that some of them might fail. The ability of a process to join or leave the system, which is very desirable in a long-lived application supposed to be highly available, is precluded in all asynchronous reliable broadcast protocols published so far.

In this paper, we introduce the first specification of a *dynamic* Byzantine reliable broadcast (DBRB) primitive that is amenable to an asynchronous implementation. The specification allows any process outside the broadcast system to join; any process that is inside the system can ask to leave. Processes inside the system can broadcast and deliver messages, whereas processes outside the system cannot. Our specification is intended for an asynchronous system for it does not require the processes to agree on the system membership. Therefore, our specification does not build on top of a group membership scheme, as does the classical *view synchrony* abstraction [10].

Our asynchronous DBRB implementation ensures that if any correct process in the system broadcasts a message, then eventually every correct process, unless it asks to leave the system, delivers that message. Moreover, if any correct process delivers a message, then every correct process, if it has not asked to leave prior to the delivery, delivers that message. The main technical difficulty addressed by our algorithm is to combine asynchrony and dynamic membership, which makes it impossible for processes to agree on the exact membership.

Two key insights enable us to face this challenge. First, starting from a known membership set at system bootstrap time, we construct a sequence of changes to this set; at any time, there is a majority of processes that record these changes. Based on this sequence, processes can determine the validity of messages. Second, before transitioning to a new membership, correct processes exchange their current state with respect to "in-flight" broadcast messages and membership changes. This prevents equivocation and conflicts.

Our algorithm assumes that, at any point in time, more than 2/3 of the processes inside the broadcast system are correct, which is tight. Moreover, we show that the "all-or-nothing" property we ensure is, in some sense, maximal. More precisely, we prove (see [14]) that in an asynchronous system, even if only one process in the system can fail, and it can merely fail by crashing, then it is impossible to implement a stronger property, ensuring that if any correct process in the system broadcasts (resp., delivers) a message, then every correct process in the system delivers that message, including those that are willing to leave.

The paper is organized as follows. In §2, we describe our system model and introduce the specification of DBRB. In §3, we overview the structure of our algorithm. In §4, we describe our implementation, and in §5, we argue its correctness. We conclude in §6 with a discussion of related and future work. Detailed proofs are delegated to the full version of the paper [14].

2 Model and Specification

We describe here our system model (§2.1) and specify our DBRB primitive (§§ 2.2 to 2.4).

2.1 A Universe of Asynchronous Processes

We consider a universe $\mathcal{U}$ of processes, subject to *Byzantine* failures: a faulty process may arbitrarily deviate from the algorithm it is assigned. Processes that are not subject to failures are *correct*. We assume an asymmetric cryptographic system. Correct processes communicate with signed messages: prior to sending a message m to a process q, a process p signs m, labeled $\langle m \rangle_{\sigma_p}$. Upon receiving the message, q can verify its authenticity and use it to prove its origin to others (non-repudiation). To simplify presentation, we omit the signature-related notation and, thus, whenever we write m, the identity of sender p and the signature are implicit and correct processes only consider messages, whether received directly or relayed by other processes, if they are equipped with valid signatures. We also use the terms "send" and "disseminate" to differentiate the points in our algorithm when a process sends a message, resp., to a single or to many destinations.

The system $\mathcal{U}$ is *asynchronous*: we make no assumptions on communication delays or relative speeds of the processes.We assume that communication is *reliable*, i.e., every message sent by a correct process to a correct process is eventually received. To describe the events that occur to an external observer and prove the correctness of the protocol, we assume a global notion of time, outside the control of the processes (not used in the protocol implementation). We consider a subset of $\mathcal{U}$ called the *broadcast system*. We discuss below how processes join or leave the broadcast system.

2.2 DBRB Interface

Our DBRB primitive exposes an interface with three operations and one callback:
1. DBRB-JOIN: used by a process outside the system to *join*.
2. DBRB-LEAVE: used by a process inside the system to *leave*.

3. DBRB-BROADCAST(m): used by a process inside the system to broadcast a message m.
4. DBRB-DELIVER(m): this callback is triggered to handle the delivery of a message m.

If a process is in the system initially, or if it has returned from the invocation of a DBRB-JOIN call, we say that it has *joined* the system. Furthermore, it is considered *participating* (or, simply, a *participant*) if it has not yet invoked DBRB-LEAVE. When the invocation of DBRB-LEAVE returns, we say that the process *leaves* the system. Note that in the interval between the invocation and the response of a DBRB-LEAVE call, the process is no longer participating, but has not yet left the system.

The following rules (illustrated in Figure 1) govern the behavior of correct processes: (i) a DBRB-JOIN operation can only be invoked if the process is not participating; moreover, we assume that DBRB-JOIN is invoked at most once; (ii) only a participating process can invoke a DBRB-BROADCAST(m) operation; (iii) a DBRB-DELIVER(m) callback can be triggered only if a process has previously joined but has not yet left the system; (iv) a DBRB-LEAVE operation can only be invoked by a participating process.

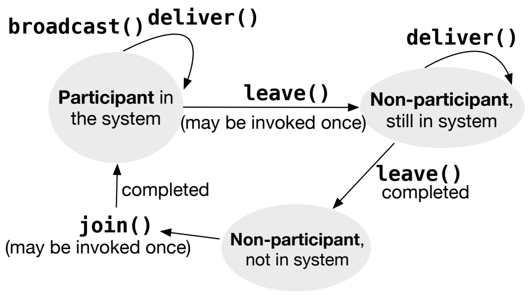

Figure 1 State transition diagram for correct processes.

2.3 Standard Assumptions

We make two standard assumptions in asynchronous reconfiguration protocols [1, 2, 5, 26], which we restate below for the sake of completeness.

▶ **Assumption 1** (Finite number of reconfiguration requests). *In every execution, the number of processes that want to join or leave the system is finite.*

▶ **Assumption 2.** *Initially, at time* 0*, the set of participants is nonempty and known to every process in* $\mathcal{U}$*.*

Assumption 1 captures the assumption that no new reconfiguration requests will be made for “sufficiently long”, thus ensuring that started operations do complete. Assumption 2 is necessary to bootstrap the system and guarantees that all processes have the same starting conditions. Additionally, we make standard cryptographic assumptions regarding the power of the adversary, namely that it cannot subvert cryptographic primitives, e.g., forge a signature.

We also assume that a weak broadcast primitive is available. The primitive guarantees that if a correct process broadcasts a message m, then every correct process eventually delivers m. In practice, such primitive can be implemented by some sort of a gossip protocol [18]. This primitive is “global” in a sense that it does not require a correct process to know all the members of $\mathcal{U}$.

2.4 Properties of DBRB

For simplicity of presentation, we assume a specific instance of DBRB in which a predefined sender process s disseminates a single message via DBRB-BROADCAST operation. The specification can easily be extended to the general case in which every participant can broadcast multiple messages, assuming that every message is uniquely identified.

▶ **Definition 1** (DBRB basic guarantees).

- Validity. *If a correct participant s broadcasts a message m at time t, then every correct process, if it is a participant at time $t' \geq t$ and never leaves the system, eventually delivers m.*
- Totality. *If a correct process p delivers a message m at time t, then every correct process, if it is a participant at time $t' \geq t$, eventually delivers m.*
- No duplication. *A message is delivered by a correct process at most once.*
- Integrity. *If some correct process delivers a message m with sender s and s is correct, then s previously broadcast m.*[1]
- Consistency. *If some correct process delivers a message m and another correct process delivers a message m', then $m = m'$.*
- Liveness. *Every operation invoked by a correct process eventually completes.*

To filter out implementations that involve *all* processes in the broadcast protocol, we add the following non-triviality property.

▶ **Definition 2** (Non-triviality). *No correct process sends any message before invoking* DBRB-JOIN *or after returning from* DBRB-LEAVE *operation.*

3 Overview

We now present the building blocks underlying our DBRB algorithm (§3.1) and describe typical scenarios: (1) a correct process joining or leaving the system (§3.2), and (2) a broadcast (§3.3).

3.1 Building Blocks

Change. We define a set of *system updates* $change = \{+, -\} \times \mathcal{U}$, where the tuple $\langle +, p\rangle$ (resp., $\langle -, p\rangle$) indicates that process p asked to join (resp., leave) the system. This abstraction captures the evolution of system membership throughout time. It is inevitable that, due to asynchrony, processes might not be able to agree on an unique system membership. In other words, two processes may concurrently consider different sets of system participants to be valid. To capture this divergence, we introduce the *view* abstraction, which defines the system membership through the lenses of some specific process at a specific point in time.

View. A view v comprises a set of updates $v.changes$. The set determines the view *membership* as $v.members = \{p \in \mathcal{U} : \langle +, p\rangle \in v.changes \wedge \langle -, p\rangle \notin v.changes\}$. For simplicity, sometimes we use $p \in v$ instead of $p \in v.members$; $|v|$ is a shorthand for $|v.members|$.

Intuitively, each correct process p in DBRB uses a view as an append-only set, to record all the *changes* that the broadcast system underwent up to a point in time, as far as p observed. Some views are "instantiated" in DBRB protocol and those views are marked as *valid* (a

[1] Recall that the identity of sender process s for a given message m is implicit in the message (§2.1).

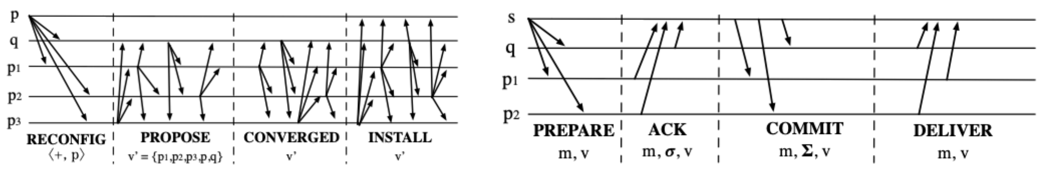

Figure 2 Protocol overview for DBRB-JOIN or DBRB-LEAVE (left), and DBRB-BROADCAST (right).

formal definition is deferred to §5). Our protocol ensures that all valid views are *comparable*. Formally, $v_1 \subset v_2$ means that $v_1.changes \subset v_2.changes$, we say that v_2 is *more recent* than v_1. Two different views are comparable if one is more recent than the other, otherwise they *conflict*. We assume that the *initial view*, i.e., the set of participants at time 0, is publicly known (Assumption 2).

A valid view v must be equipped with a *quorum system*: a collection of subsets of $v.members$. We choose the quorums to be all subsets of size $v.q = |v| - \lfloor \frac{|v|-1}{3} \rfloor$.

▶ **Assumption 3** (Quorum systems). *In every valid view v, the number of Byzantine processes is less than or equal to $\lfloor \frac{|v|-1}{3} \rfloor$ and at least one quorum in v contains only correct processes.*

Thus, every two quorums of a valid view have a correct process in common and at least one quorum contains only correct processes.[2]

Sequence of views. We now build upon the comparability of views to obtain the abstraction of a *sequence of views*, or just *sequence*. A sequence *seq* is a set of mutually comparable views. Note that a set with just one view is, trivially, a sequence of views, so is the empty set.

Reliable multicast. In addition to the use of signed messages (§2.1), we build our algorithm on top of an elementary (static) reliable Byzantine broadcast protocol. We instantiate this protocol from a standard solution in the literature for a static set of processes, as described e.g., in [9]. The terms "*R-multicast*" and "*R-delivery*" refer to the request to broadcast a message and deliver a message via this protocol to (or from) a static set of processes. For completeness, we provide the pseudocode of the static reliable Byzantine broadcast primitive in [14].

3.2 DBRB-JOIN and DBRB-LEAVE Operations

Upon invoking the DBRB-JOIN operation, a process p first learns the current membership – i.e., the most recent view v – of the broadcast system through a *View Discovery* protocol (§4.1). The joining operation then consists of four steps (the left part of Figure 2). First, p disseminates a ⟨RECONFIG, $\langle +, p \rangle$⟩ message to members of v. In the second step, when any correct process q from v receives the RECONFIG message, q proposes to change the system membership to a view v', where v' is an extension of v including the change $\langle +, p \rangle$. To do so, q disseminates to members of v a PROPOSE message, containing the details of v'. Third, any other correct member in v waits until $v.q$ matching PROPOSE messages (a quorum

[2] Note that this bound applies both to processes that are active participants, as well as processes leaving the system. This requirement can be relaxed in practice by enforcing a correct process that leaves the system to destroy its private key. Even if the process is later compromised, it will not be able to send any protocol messages. Note that we assume that messages sent while the process was correct cannot be withdrawn or modified.

of v confirms the new view). Once a process collects the confirmation, it disseminates a $\langle$CONVERGED, $v'\rangle$ message to members of v. This concludes step three. In the fourth step, each correct process q in v waits to gather matching CONVERGED messages from a quorum (i.e., $v.q$) of processes. We say that processes that are members of view v are trying to *converge* on a new membership. Then, q triggers an *R-multicast* of the $\langle$INSTALL, $v'\rangle$ message to members of $v \cup v'$; recall that the process p belongs to v'. Upon *R-delivery* of an INSTALL message for v', any process q updates its current view to v'. The DBRB-JOIN operation finishes at process p once this process receives the INSTALL message for $v' \ni p$. From this instant on, p is a participant in the system.

The steps executed after a correct process p invokes the DBRB-LEAVE operation are almost identical, except for the fact that p still executes its "duties" in DBRB until DBRB-LEAVE returns.[3]

3.3 DBRB-BROADCAST Operation

A correct process s that invokes DBRB-BROADCAST(m) first disseminates a PREPARE message to every member of the s' current view v. When a correct process q receives this message, q sends an ACK message to s, representing a signed statement asserting that q indeed received m from s. Once s collects a quorum of matching ACK messages for m, s constructs a *message certificate* Σ out of the collected signatures ρ, and disseminates this certificate to every member of v as part of a COMMIT message. When any correct process q receives a COMMIT message with a valid certificate for m for the first time, q relays this message to all members of view v. Moreover, q sends a DELIVER message to the sender of the COMMIT message. Once any process q collects a quorum of matching DELIVER messages, q triggers DBRB-DELIVER(m). The right part of Figure 2 presents the overview of this operation. In Figure 2, we depict process s collecting enough DELIVER messages to deliver m, assuming that all processes in the system use the same view. The details of how views are changed during an execution of a broadcast operation are given in §4.2.

4 DBRB Algorithm

In this section, we describe our DBRB algorithm, starting with dynamic membership (§4.1), and continuing with broadcast (§4.2). We also present an illustrative execution of DBRB (§4.3).

Algorithm 1 introduces the variables that each process p maintains, as well as two helper functions to compute the least recent and most recent view of a given sequence, respectively.

4.1 Dynamic Membership

Algorithms 2 and 3 contain the pseudocode of the DBRB-JOIN and DBRB-LEAVE operations. Let us first discuss the join operation.

After a correct process p invokes the DBRB-JOIN operation, p obtains the most recent view of the system, and it does so through the View Discovery protocol. We describe the View Discovery protocol at the end of this section; for the moment it suffices to say that p obtains the most recent view v and updates its local variable cv to reflect this view. Next, process p disseminates a $\langle$RECONFIG, $\langle +, p\rangle$, $cv\rangle$ message to every member of cv (Algorithm 2)

[3] There is a detail we deliberately omitted from this high-level description and we defer to §4.1: multiple processes may try to join the system concurrently, and thereby multiple PROPOSE messages may circulate at the same time. These messages comprise different views, e.g., one could be for a view v' and another for v''. These conflicts are unavoidable in asynchronous networks. For this reason, PROPOSE messages (and other protocol messages) operate at the granularity of sequences, not individual views. If conflicts occur, sequences support union and ordering, allowing reconciliation of v' with v'' on a sequence that comprises their union.

notifying members of cv of its intention to join. The view discovery and the dissemination are repeated until the *joinComplete* event triggers or a quorum of confirmation messages has been collected for some view v to which RECONFIG message was broadcast (Algorithm 2).

Every correct member r of the view cv proposes a new system membership that includes process p, once r receives the aforementioned RECONFIG message from process p. The new proposal is incorporated within a *sequence* of views SEQ^v, $v = cv$, (containing, initially, just one view) and disseminated to all members of the view cv via a PROPOSE message (Algorithm 2).

The leaving operation invocation is similar: Process p disseminates a RECONFIG message with $\langle -, p\rangle$ as an argument, and process r proposes a new system membership that *does not* include p. The main difference with the joining operation is that if p delivered or is the sender of a message, p must ensure validity and totality properties of DBRB before disseminating a RECONFIG message (Algorithm 2).

Let us now explain how a new view is installed in the system. The correct process $r \in cv$ receives PROPOSE messages disseminated by other members of cv. First, r checks whether it *accepts*[4] the received proposal (recall that a proposal is a sequence of views). Moreover, r checks whether the received proposed sequence *seq* is well-formed, i.e., whether *seq* satisfies the following: (1) *seq* is a sequence of views, (2) there is at least one view in *seq* that r is not aware of, and (3) every view in *seq* is more recent than cv.

If all the checks have passed, the process r uses the received PROPOSE message to update its own proposal. This is done according to two cases:

1. There are conflicts between r's and the received proposal (Algorithm 2 to Algorithm 2). In this case, r creates a new proposal containing r's last converged sequence for the view[5] and a new view representing the union of the most recent views of two proposals.
2. There are no conflicts (Algorithm 2). In this case, r executes the union of its previous and received proposal in order to create a new proposal.

Once r receives the same proposal from a quorum of processes, r updates its last converged sequence (Algorithm 2) and disseminates it within a CONVERGED message (Algorithm 2).

When r receives a CONVERGED message for some sequence of views seq' and some view v (usually v is equal to the current view cv of process r, but it could also be a less recent view than cv) from a quorum of members of the view v (Algorithm 2), r creates and reliably disseminates an INSTALL message that specifies the view that should be replaced (i.e., v), the least recent view of the sequence seq' denoted by ω (Algorithm 2) and the entire sequence seq' (Algorithm 2). Moreover, we say that seq' is *converged on* to replace v. An INSTALL message is disseminated to processes that are members of views v or ω (Algorithm 2). Note that INSTALL messages include a quorum of signed CONVERGED messages which ensures its authenticity (omitted in Algorithms 2 and 3 for brevity).

Once the correct process r receives the INSTALL message (Algorithm 3), r enters the installation procedure in order to update its current view of the system. There are four parts to consider:

1. Process r was a member of a view v (Algorithm 3): Firstly, r checks whether $cv \subset \omega$, where cv is the current view of r. If this is the case, r stops processing PREPARE, COMMIT

[4] Process r accepts a sequence of views *seq* to replace a view v if $seq \in FORMAT^v$ or $\emptyset \in FORMAT^v$ (Algorithm 2). The following holds at every correct process that is a member of the initial view of the system v_0: $\emptyset \in FORMAT^{v_0}$. Note that $FORMAT^v$, for any view v, is a set of sequences, i.e., a set of sets.

[5] We say that *seq* is the last converged sequence for a view v of a process if the process receives the same proposal to replace the view v from a quorum of members of v (variable $LCSEQ^v$).

Algorithm 1 DBRB algorithm: local variables of process p and helper functions.

```
1:  variables:
2:      $cv = v_0$                                   // current view; $v_0$ is the initial view
3:      $RECV = \emptyset$                           // set of pending updates (i.e., join or leave)
4:      $SEQ^v = \emptyset$                          // set of proposed sequences to replace $v$
5:      $LCSEQ^v = \emptyset$                        // last converged sequence to replace $v$
6:      $FORMAT^v = \emptyset$                       // replacement sequence for view $v$
7:      $cer = \bot$                                 // message certificate for $m$
8:      $v_{cer} = \bot$                             // view in which certificate is collected
9:      ▷ set of messages allowed to be acknowledged; initially, any message could be acknowledged by a process
10:     $allowed\_ack = \bot$                        // $\bot$ - any message, $\top$ - no message
11:     $stored = false$; $stored\_value = \bot$
12:     $can\_leave = false$                         // process is allowed to leave
13:     $delivered = false$                          // $m$ delivered or not
14:     ▷ for every process $q \in \mathcal{U}$ and every valid view $v$
15:     $acks[q, v] = \bot$; $\Sigma[q, v] = \bot$; $deliver[q, v] = \bot$
16:     $State = \bot$                               // state of the process; consists of ack, conflicting and stored fields

17: function $least\_recent(seq)$ returns $\omega \in seq : \nexists \omega' \in seq : \omega' \subset \omega$
18: function $most\_recent(seq)$ returns $\omega \in seq : \nexists \omega' \in seq : \omega \subset \omega'$
```

and RECONFIG messages (Algorithm 3; see §4.2). Therefore, process r will not send any ACK or DELIVER message for PREPARE or COMMIT messages associated with v (and views preceding v). The same holds for RECONFIG messages. We refer to acknowledged and stored messages by a process as the *state* of the process (represented by the *State* variable). The fact that r stops processing the aforementioned messages is important because r needs to convey this information via the STATE-UPDATE message (Algorithm 3) to the members of the new view ω. Therefore, a conveyed information is "complete" since a correct process r will never process any PREPARE, COMMIT or RECONFIG message associated with "stale" views (see §4.2).

2. View ω is more recent than r's current view cv (Algorithm 3 to Algorithm 3): Process r waits for $v.q$ of STATE-UPDATE messages (Algorithm 3) and processes received states (Algorithm 3). STATE-UPDATE messages carry information about: (1) a message process is allowed to acknowledge (*allowed_ack* variable), (2) a message stored by a process (*stored_value* variable), and (3) reconfiguration requests observed by a process (see §4.2). Hence, a STATE-UPDATE message contains at most two PREPARE messages associated with some view and properly signed by s (corresponds to (1)). Two PREPARE messages are needed if a process observes that s broadcast two messages and are used to convince other processes not to acknowledge any messages (variable $State.conflicting$; Algorithm 4). Moreover, STATE-UPDATE messages contain at most one COMMIT message associated with some view with a valid message certificate (variable $State.stored$) and properly signed by s (corresponds to (2)), and a (possibly empty) list of properly signed RECONFIG messages associated with some installed view (corresponds to (3)). Note that processes include only PREPARE, COMMIT and RECONFIG messages associated with some view $v'' \subseteq v$ in the STATE-UPDATE message they send (incorporated in the $state(v)$ function). The reason is that processes receiving these STATE-UPDATE messages may not know whether views $v'' \supset v$ are indeed "created" by our protocol and not "planted" by faulty processes.
3. Process r is a member of $\omega \supset cv$ (Algorithm 3 to Algorithm 3): If this is the case, r updates its current view (Algorithm 3). Moreover, r *installs* the (updated) current view cv if the sequence received in the INSTALL message does not contain other views that are more recent than cv (Algorithm 3).
4. Process r is not a member of $\omega \supset cv$ (Algorithm 3 to Algorithm 3): A leaving process r executes the View Discovery protocol (Algorithm 3) in order to ensure totality of DBRB (we explain this in details in §4.2). When r has "fulfilled" its role in ensuring totality of DBRB, r leaves the system (Algorithm 3).

Algorithm 2 DBRB-JOIN and DBRB-LEAVE implementations at process p.

```
19: procedure DBRB-JOIN()
20:     repeat
21:         $cv$ = view_discovery($cv$)
22:         disseminate ⟨RECONFIG, $\langle +, p\rangle, cv$⟩ to all $q \in cv.members$
23:     until $joinComplete$ is triggered or $v.q$ ⟨REC-CONFIRM, $v$⟩ messages collected for some $v$
24:     wait for $joinComplete$ to be triggered

25: procedure DBRB-LEAVE()
26:     if $delivered \lor p = s$ then wait until $can\_leave$
27:     repeat in each installed view $cv$ do            // in each subsequent view $p$ installs
28:         disseminate ⟨RECONFIG, $\langle -, p\rangle, cv$⟩ to all $q \in cv.members$
29:     until $leaveComplete$ is triggered or $v.q$ ⟨REC-CONFIRM, $v$⟩ messages collected for some $v$
30:     wait for $leaveComplete$ to be triggered

31: upon receipt of ⟨RECONFIG, $\langle c, q\rangle, v$⟩ from $q$                // $c \in \{-, +\}$
32:     if $v = cv \land \langle c, q\rangle \notin v \land$ (if $(c = -)$ then $\langle +, q\rangle \in v$) then
33:         $RECV = RECV \cup \{\langle c, q\rangle\}$
34:         send ⟨REC-CONFIRM, $cv$⟩ to $q$
35:     end if

36: upon $RECV \neq \emptyset \land installed(cv)$ do
37:     if $SEQ^{cv} = \emptyset$ then
38:         $SEQ^{cv} = \{cv \cup RECV\}$
39:         disseminate ⟨PROPOSE, $SEQ^{cv}, cv$⟩ to all $q \in cv.members$
40:     end if

41: upon receipt of ⟨PROPOSE, $seq, v$⟩ from $q \in v.members$ such that $seq \in FORMAT^v \lor \emptyset \in$
    $FORMAT^v$
42:     if valid($seq$) then                                 // filter incorrect proposals
43:         if conflicting($seq, SEQ^v$) then
44:             $\omega = most\_recent(seq)$
45:             $\omega' = most\_recent(SEQ^v)$
46:             ▷ merge the last view from the local and $q$'s proposal
47:             $SEQ^v = LCSEQ^v \cup \{\omega \cup \omega'\}$
48:         else                                  // no conflicts, just merge the proposals
49:             $SEQ^v = SEQ^v \cup seq$
50:         end if
51:         disseminate ⟨PROPOSE, $SEQ^v, v$⟩ to all $q' \in v.members$
52:     end if

53: upon receipt of ⟨PROPOSE, $SEQ^v, v$⟩ from $v.q$ processes in $v$
54:     $LCSEQ^v = SEQ^v$
55:     disseminate ⟨CONVERGED, $SEQ^v, v$⟩ to all $q \in v.members$

56: upon receipt of ⟨CONVERGED, $seq', v$⟩ from $v.q$ processes in $v$
57:     $\omega = least\_recent(seq')$
58:     $R\text{-}multicast(\{j : j \in v.members \lor j \in \omega.members\}$, ⟨INSTALL, $\omega, seq', v$⟩)
```

View Discovery. Views "created" during an execution of DBRB form a sequence (see [14]). The View Discovery subprotocol provides information about the sequence of views incorporated in an execution so far. Since every correct process in the system knows the initial view (Assumption 2) and valid transition between views implies the existence of an INSTALL message with a quorum of properly signed CONVERGED messages, any sequence of views starting from the initial view of the system such that appropriate INSTALL messages "connect" adjacent views can be trusted.

A correct process that has invoked the DBRB-JOIN operation and has not left the system executes the View Discovery subprotocol constantly. Once a correct process starts trusting a sequence of views, it disseminates that information to all processes in the universe. A correct process executing the View Discovery subprotocol learns which sequences of views are trusted by other processes. Once the process observes a sequence of views allegedly trusted by a process, it can check whether the sequence is properly formed (as explained above) and if that is the case, the process can start trusting the sequence and views incorporated in it (captured by the view_discovery function for the joining and leaving process; Algorithms 2 and 3).

Algorithm 3 DBRB algorithm: installing a view at process p.

```
59: upon $R\text{-}delivery(\{j : j \in v.members \vee j \in \omega.members\}, \langle \text{INSTALL}, \omega, seq, v\rangle)$ do
60:     $FORMAT^{\omega} = FORMAT^{\omega} \cup \{seq \setminus \{\omega\}\}$
61:     if $p \in v.members$ then                                        // $p$ was a member of $v$
62:         if $cv \subset \omega$ then stop processing PREPARE, COMMIT and RECONFIG messages
63:         $R\text{-}multicast(\{j : j \in v.members \vee j \in \omega.members\}, \langle \text{STATE-UPDATE}, state(v), RECV\rangle)$
64:     end if
65:     if $cv \subset \omega$ then                              // $\omega$ is more recent than $p$'s current view
66:         wait for $\langle \text{STATE-UPDATE}, *, *\rangle$ messages from $v.q$ processes in $v$  // from the reliable broadcast
67:         $req = \{$reconfiguration requests from STATE-UPDATE messages$\}$
68:         $RECV = RECV \cup (req \setminus \omega.changes)$
69:         $states = \{$states from STATE-UPDATE messages$\}$
70:         $installed(\omega) = false$
71:         invoke $state\text{-}transfer(states)$                                  // Algorithm 4
72:         if $p \in \omega.members$ then                                       // $p$ is in $\omega$
73:             $cv = \omega$
74:             if $p \notin v.members$ then trigger $joinComplete$            // can return from DBRB-JOIN
75:             if $\exists \omega' \in seq : cv \subset \omega'$ then
76:                 $seq' = \{\omega' \in seq : cv \subset \omega'\}$
77:                 if $SEQ^{cv} = \emptyset \wedge \forall \omega \in seq' : cv \subset \omega$ then
78:                     $SEQ^{cv} = seq'$
79:                     disseminate $\langle \text{PROPOSE}, SEQ^{cv}, cv\rangle$ to all $q \in cv.members$
80:                 end if
81:             else
82:                 $installed(cv) = true$
83:                 resume processing PREPARE, COMMIT and RECONFIG messages
84:                 invoke $new\text{-}view()$                                   // Algorithm 4
85:             end if
86:         else                                                  // $p$ is leaving the system
87:             if $stored$ then
88:                 while $\neg can\_leave$ do
89:                     $cv = \text{view\_discovery}(cv)$
90:                     disseminate $\langle \text{COMMIT}, m, cer, v_{cer}, cv\rangle$ to all $q \in cv.members$
91:                 end while
92:             end if
93:             trigger $leaveComplete$                              // can return from DBRB-LEAVE
94:         end if
95:     end if
```

The View Discovery protocol addresses two main difficulties: (1) it enables processes joining and leaving the system to learn about the current membership of the system, (2) it is crucial to ensure the consistency, validity and totality properties of DBRB since it supplies information about views "instantiated" by the protocol and associated quorum systems. We formally discuss the View Discovery protocol in the full version of the paper [14].

4.2 Broadcast

In order to broadcast some message m, processes in DBRB use the following types of messages:
PREPARE: When a correct process s invokes a DBRB-BROADCAST(m) operation, the algorithm creates a $m_{prepare} = \langle \text{PREPARE}, m, cv_s\rangle$ message, where cv_s is the current view of the system of process s. Message $m_{prepare}$ is sent to every process that is a member of cv_s (Algorithm 4). Process s disseminates the PREPARE message if cv_s is installed by s; otherwise, s does not disseminate the message to members of cv_s (Algorithm 4), but rather waits to install some view and then disseminates the PREPARE message (Algorithm 4).
ACK: When a correct process q receives $m_{prepare}$ message, q firstly checks whether view specified in $m_{prepare}$ is equal to the current view of q (Algorithm 4). If that is the case, q checks whether it is allowed to send an ACK message for m (see Consistency paragraph in §5; Algorithm 4) and if it is, q sends $m_{ack} = \langle \text{ACK}, m, \sigma, cv_q\rangle$ message to process s (i.e., the sender of $m_{prepare}$), where σ represents the signed statement that s sent m to q (Algorithm 4).

Algorithm 4 DBRB-BROADCAST(m) and DBRB-DELIVER(m) implementations at process p.

```
96: procedure $state\text{-}transfer(states)$
97:     if $(allowed\_ack = \bot \vee allowed\_ack = m) \wedge m$ is the only acknowledged message among $states$ then
98:         $allowed\_ack = m$; $update\_if\_bot(State.ack, prepare\_msg)$        // updated only if it is $\bot$
99:     else if there exist at least two different messages acknowledged among $states$ then
100:        ▷ $p$ and $p'$ are different PREPARE messages
101:        $allowed\_ack = \top$; $update\_if\_bot(State.conflicting, p, p')$; $State.ack = \bot$
102:    else if there exists a state among $states$ such that it provides two different broadcast messages then
103:        $allowed\_ack = \top$; $update\_if\_bot(State.conflicting, p, p')$; $State.ack = \bot$
104:    end if
105:    if $\neg stored \wedge$ there exists a stored message $m$ with a valid message certificate among $states$ then
106:        $stored = true$; $stored\_value = (m, cer, v_{cer})$ // $cer$ is the message certificate collected in view $v_{cer}$
107:        $update\_if\_bot(State.stored, commit\_msg)$        // updated only if it is $\bot$
108:    end if

109: procedure $new\text{-}view()$
110:    if $p = s \wedge cer = \bot$ then disseminate $\langle$PREPARE, $m, cv\rangle$ to all $q \in cv.members$
111:    if $p = s \wedge cer \neq \bot \wedge \neg can\_leave$ then disseminate $\langle$COMMIT, $m, cer, v_{cer}, cv\rangle$ to all $q \in cv.members$
112:    if $p \neq s \wedge stored \wedge \neg can\_leave$ then disseminate $\langle$COMMIT, $m, cer, v_{cer}, cv\rangle$ to all $q \in cv.members$

113: procedure DBRB-BROADCAST$(m)$
114:    if $installed(cv)$ then disseminate $\langle$PREPARE, $m, cv\rangle$ to all $q \in cv.members$

115: upon receipt of $\langle$PREPARE, $m, v\rangle$ from $s \in v.members$ such that $v = cv$
116:    if $allowed\_ack = m \vee allowed\_ack = \bot$ then
117:        $allowed\_ack = m$; $update\_if\_bot(State.ack, \langle$PREPARE, $m, v\rangle)$        // updated only if it is $\bot$
118:        $\sigma = sign(m, cv)$; send $\langle$ACK, $m, \sigma, cv\rangle$ to $s$
119:    end if

120: upon receipt of $\langle$ACK, $m, \sigma, v\rangle$ from $q \in v.members$        // only process $s$
121:    if $acks[q, v] = \bot \wedge verifysig(q, m, v, \sigma)$ then $acks[q, v] = m$; $\Sigma[q, v] = \sigma$

122: upon exists $m \neq \bot$ and $v$ such that $|\{q \in v.members | acks[q, v] = m\}| \geq v.q \wedge cer = \bot$ do
123:    $cer = \{\Sigma[q, v] : acks[q, v] = m\}$; $v_{cer} = v$
124:    if $installed(cv)$ then disseminate $\langle$COMMIT, $m, cer, v_{cer}, cv\rangle$ to all $q' \in cv.members$

125: upon receipt of $\langle$COMMIT, $m, cer, v_{cer}, v\rangle$ from $q$ such that $v = cv$
126:    if $verify\_certificate(cer, v_{cer}, m)$ then
127:        if $\neg stored$ then
128:            $stored = true$; $stored\_value = (m, cer, v_{cer})$
129:            $update\_if\_bot(State.stored, \langle$COMMIT, $m, cer, v_{cer}, v\rangle)$        // updated only if it is $\bot$
130:            disseminate $\langle$COMMIT, $m, cer, v_{cer}, cv\rangle$ to all $q' \in cv.members$
131:        end if
132:        send $\langle$DELIVER, $m, cv\rangle$ to $q$
133:    end if

134: upon receipt of $\langle$DELIVER, $m, v\rangle$ from $q \in v.members$
135:    if $deliver[q, v] = \bot$ then $deliver[q, v] = \top$

136: upon exists $v$ such that $|\{q \in v.members | deliver[q, v] = \top\}| \geq v.q$ for the first time do
137:    $delivered = true$
138:    invoke DBRB-DELIVER$(m)$
139:    $can\_leave = true$        // If $p = s$, DBRB-BROADCAST is completed
```

When some process q sends an ACK message for m (m is a second argument of the message), we say that q *acknowledges* m. Moreover, if an ACK message is associated with some view v, we say that q acknowledges m in a view v.

COMMIT: When process s receives a quorum of appropriate ACK messages associated with the same view v for m (Algorithm 4), s collects received signed statements into a *message certificate*. Process s then creates $m_{commit} = \langle$COMMIT, m, cer, v_{cer}, $cv_s\rangle$ message and disseminates m_{commit} to every process that is a member of cv_s (Algorithm 4). Note that cv_s may be different from v (we account for this in the rest of the section). Moreover, s disseminates the COMMIT message (Algorithm 4) if cv_s is installed by s; otherwise, s does not disseminate the message to members of cv_s, but rather waits to install some view and then disseminates the COMMIT message (Algorithm 4).

DELIVER: When a correct process q receives m_{commit} message, it firstly checks whether view specified in m_{commit} is equal to the current view of q (Algorithm 4). If that is the case and the message certificate is valid (Algorithm 4), q "stores" m (Algorithm 4) and sends $m_{deliver} = \langle \text{DELIVER}, m, cv_q \rangle$ to process s as a an approval that s can deliver m (Algorithm 4). When a process q executes Algorithm 4 or Algorithm 4 for a message m, we say that q *stores* m. Observe that q also disseminates m_{commit} in order to deliver m itself (Algorithm 4).

Lastly, once a correct process receives a quorum of appropriate DELIVER messages associated with the same view v for m (Algorithm 4), it delivers m (Algorithm 4).

Every PREPARE, ACK, COMMIT and DELIVER message is associated with one specific view.

We can divide the broadcasting of message m by the correct sender s into two phases:

- **Certificate collection phase**: This phase includes a dissemination of an appropriate PREPARE message and a wait for a quorum of ACK messages by process s. Note that PREPARE and ACK messages are associated with the same view v. We say that certificate collection phase is *executed* in view v. Moreover, if s indeed receives a quorum of ACK messages associated with v, we say that certificate collection phase is *successfully executed* in v. In that case, sometimes we say that s collects a message certificate in v.
- **Storing phase**: In this phase, each correct process p (including s) disseminates a COMMIT message (containing a valid message certificate collected in the previous phase), and waits for a quorum of DELIVER messages. Note that COMMIT and DELIVER messages are associated with the same view v. We say that storing phase is *executed* in view v. Moreover, if p indeed receives a quorum of DELIVER messages associated with v, we say that storing phase is *successfully executed* in v.

Observe that the certificate collection phase can be successfully executed in some view v, whereas the storing phase can be executed in some view $v' \supset v$. This is the reason why we include v_{cer} argument in a COMMIT message, representing the view in which a message certificate is collected. Lastly, in order to ensure validity and totality, processes must disseminate PREPARE and COMMIT messages in new views they install until they collect enough ACK and DELIVER messages, respectively. This mechanism is captured in the *new-view* procedure (Algorithm 4) that is invoked when a view is installed (Algorithm 3).

4.3 Illustration

Consider four participants at time $t = 0$. Process p_1 broadcasts a message m. Hence, p_1 sends to processes p_1, p_2, p_3, p_4 a $m_{prepare} = \langle \text{PREPARE}, m, v_0 \rangle$ message, where $v_0 = \{\langle +, p_1 \rangle, \langle +, p_2 \rangle, \langle +, p_3 \rangle, \langle +, p_4 \rangle\}$. Since all processes consider v_0 as their current view of the system at time of receiving of $m_{prepare}$ message, they send to p_1 an appropriate ACK message and p_1 collects a quorum (with respect to v_0) of ACK messages for m.

However, process p_5 invokes a DBRB-JOIN operation and processes p_2, p_3, p_4, p_5 set $v_1 = \{\langle +, p_1 \rangle, \langle +, p_2 \rangle, \langle +, p_3 \rangle, \langle +, p_4 \rangle, \langle +, p_5 \rangle\}$ as their current view of the system. Process p_1 still considers v_0 as its current view of the system and disseminates a $m_{commit} = \langle \text{COMMIT}, m, cer, v_{cer} = v_0, v_0 \rangle$ message. Processes p_2, p_3 and p_4 do not store m, since v_0 (specified in m_{commit} message) is not their current view. Observe that p_1 stores m since v_0 is still the current view of the system from p_1's perspective.

Once process p_1 assigns v_1 as its current view of the system, it disseminates $m_{commit} = \langle \text{COMMIT}, m, cer, v_{cer} = v_0, v_1 \rangle$ message to processes that are members of v_1 and they all store m and relay m_{commit} message to all processes that are members of v_1. Hence, they all deliver message m once they collect a quorum (with respect to v_1) of matching DELIVER messages. In this execution p_1 has successfully executed a certificate collection phase in v_0 and then reused the message certificate to relay an appropriate COMMIT message to processes that are members of v_1 (since the system has reconfigured to v_1). Note that Figure 3 depicts the described execution. For presentation simplicity, Figure 3 just shows the COMMIT and DELIVER messages that allow process p_1 to deliver m.

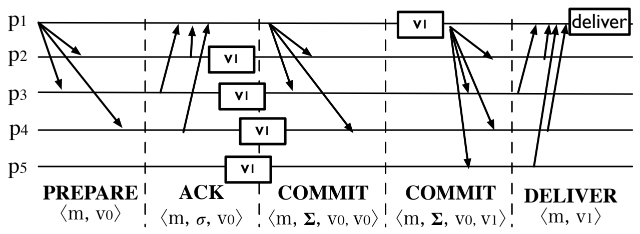

Figure 3 Example of a broadcast operation in DBRB algorithm, considering a dynamic membership.

5 DBRB Algorithm Correctness

We now give an intuition of why our DBRB algorithm is correct; we give formal arguments in the full version of the paper [14].

We first define the notions of *valid* and *installed* views. A view v is *valid* if: (1) v is the initial view of the system, or (2) a sequence $seq = v \to \ldots$ is converged on to replace some valid view v'. A valid view v is *installed* if a correct process $p \in v$ processed PREPARE, COMMIT and RECONFIG messages associated with v during an execution. By default, the initial view of the system is installed. Lastly, our implementation ensures that installed views form a sequence of views.

Liveness. DBRB-JOIN and DBRB-LEAVE operations complete because any change "noticed" by a quorum of processes is eventually processed. Intuitively, a sequence can be converged on if a quorum of processes propose that sequence. Moreover, noticed changes are transferred to new valid views. DBRB-BROADCAST operation completes since a correct sender eventually collects a quorum of DELIVER messages associated with an installed view (see the next paragraph).

Validity. Recall that we assume a finite number of reconfiguration requests in any execution of DBRB (Assumption 1), which means that there exists a view v_{final} from which the system will not be reconfigured. In order to prove validity, it suffices to show that every correct member of v_{final} delivers a broadcast message.

A correct process s that broadcasts a message m executes a certificate collection phase in some installed view v (the current view of s). Even if s does not successfully execute a certificate collection phase in views that precede v_{final} in the sequence of installed views, s successfully executes a certificate collection phase in v_{final}. Note that process s does not leave the system before it collects enough DELIVER messages (ensured by the check at Algorithm 2 and the assignment at Algorithm 4).

Moreover, a correct process p that stored a message m eventually collects a quorum of DELIVER messages associated with some installed view v. As in the argument above, even if p does not collect a quorum of DELIVER messages associated with views that precede v_{final}

in the sequence of installed views, p does that in v_{final}. Observe that if at least a quorum of processes that are members of some installed view v store a message m, then every correct process $p \in v'$, where view $v' \supseteq v$ is installed, stores m. Let us give the intuition behind this claim. Suppose that view v' directly succeeds view v in the sequence of views installed in the system. Process $p \in v'$ waits for states from at least a quorum of processes that were members of v (Algorithm 3) before it updates its current view to v'. Hence, p receives from at least one process that m is stored and then p stores m (Algorithm 4). The same holds for the correct members of v.

It now suffices to show that s collects a quorum (with respect to some installed view) of confirmations that m is stored, i.e., DELIVER messages. Even if the correct sender does not collect a quorum of DELIVER messages in views that precede v_{final}, it collects the quorum when disseminating the COMMIT message to members of v_{final}. Suppose now that the sender collects the aforementioned quorum of DELIVER messages in some installed view v. If $v \neq v_{final}$, every correct member of v_{final} stores and delivers m (because of the previous argument). If $v = v_{final}$, the reliable communication and the fact that the system can not be further reconfigured guarantee that every correct member of v_{final} stores and, thus, delivers m.

Totality. The intuition here is similar to that behind ensuring validity. Consider a correct process p that delivers a message m: p successfully executed a storing phase in some installed view v. This means that every member of an installed view $v' \supseteq v$ stores m. Consider a correct participant q that expressed its will to leave after process p had delivered m. This implies that $q \in v''$, where $v'' \supseteq v$ is an installed view, which means that process q eventually stores m. As in the previous paragraph, we conclude that process q eventually collects enough DELIVER messages associated with some installed view and delivers m.

Consistency. A correct process delivers a message only if there exists a message certificate associated with the message (the check at Algorithm 4). Hence, the malicious sender s must collect message certificates for two different messages in order for the consistency to be violated.

Suppose that process s has successfully executed a certificate collection phase in some installed view v for a message m. Because of the quorum intersection and the verification at Algorithm 4, it is impossible for s to collect a valid message certificate in v for some message $m' \neq m$. Consider now an installed view v' that directly succeeds view v in the sequence of installed views. Since s collected a message certificate for m in v, every correct process $p \in v'$ receives from at least one process from the view v that it is allowed to acknowledge only message m (Algorithm 4). It is easy to see that this holds for every installed view $v'' \supset v'$. Therefore, if s also collects a message certificate for some message m', then $m' = m$ and the consistency holds.

No duplication. Trivially follows from Algorithm 4.

Integrity. Consider a correct process q that delivers a message m. There is a message certificate for m collected in some installed view v by s. A message certificate for m is collected since a quorum of processes in v have sent an appropriate ACK message for m. A correct process sends an ACK message only when it receives an appropriate PREPARE message. Consequently, message m was broadcast by s.

6 Related Work & Conclusions

DBRB vs. Static Byzantine Reliable Broadcast. Our DBRB abstraction generalizes *static* BRB (Byzantine reliable broadcast [8, 22]). Assuming that no process joins or leaves the system, the two abstractions coincide. In a dynamic setting, the validity property of DBRB stipulates that only processes that do not leave the system deliver the appropriate messages. Moreover, the totality property guarantees that only processes that have not expressed their will to leave deliver the message. We prove that stronger variants of these properties are impossible in our model.

Passive and Active Reconfiguration. Some reconfigurable systems [6, 3, 4] assume that processes join and leave the system under a specific *churn* model. Intuitively, the consistency properties of the implemented service, e.g., an atomic storage, are ensured assuming that the system does not evolve too quickly and there is always a certain fraction of correct members in the system. In DBRB, we model this through the quorum system assumption on valid views (Assumption 3). Our system model also assumes that booting finishes by time 0 (Assumption 2), thus avoiding the problem of unbounded booting times which could be problematic in asynchronous network [27].

Active reconfiguration allows the processes to explicitly propose configuration updates, e.g., sets of new process members. In *DynaStore* [1], reconfigurable dynamic atomic storage is implemented in an asynchronous environment (i.e., without relying on consensus). Dynastore implicitly generates a graph of views which provides a way of identifying a sequence of views in which clients need to execute their r/w operations. SpSn [12] proposes to capture this order via the *speculating snapshot* algorithm (SpSn). SmartMerge [17] implements a reconfigurable storage in which not only system membership but also its quorum system can be reconfigured, assuming that a *static* external lattice agreement is available. In [20], it was shown that *reconfigurable lattice agreement* can get rid of this assumption and still implement a large variety of reconfigurable objects. The approach was then extended to the Byzantine fault model [21]. FreeStore [2] introduced *view generator*, an abstraction that captures the agreement demands of reconfiguration protocols. Our work is highly inspired by FreeStore, which algorithmic and theoretical approach we adapt to an arbitrary failure model.

All reconfigurable solutions discussed above were applied exclusively to shared-memory emulations. Moreover, most of them assumed the crash fault model. In contrast, in this paper, we address the problem of dynamic *reliable broadcast*, assuming an arbitrary (Byzantine) failure model. Also, we do not distinguish between clients and replicas, and assume that every process can only suggest itself as a candidate to join or leave the system. Unlike the concurrent work by Kumar and Welch on Byzantine-tolerant registers [19], our solution can tolerate unbounded number of Byzantine failures, as long as basic quorum assumptions on valid views are maintained.

Broadcast Applications. Reliable broadcast is one of the most pervasive primitives in distributed applications [25]. For instance, broadcast can be used for maintaining caches in cloud services [13], or in a publish-subscribe network [11]. Even more interestingly, Byzantine fault-tolerant reliable broadcast (e.g., dynamic solution such as our DBRB, as well as static solutions [8, 16, 23]) are sufficiently strong for implementing decentralized online payments, i.e., cryptocurrencies [15].

Summary. This paper presents the specification of DBRB (dynamic Byzantine reliable broadcast), as well as an asynchronous algorithm implementing this primitive. DBRB generalizes traditional Byzantine reliable broadcast, which operates in static environments, to work in a dynamic network. To the best of our knowledge, we are the first to investigate an arbitrary failure model in implementing dynamic broadcast systems. The main merit of our approach is that we did not rely on a consensus building blocks, i.e., DBRB can be implemented completely asynchronously.

References

1 Marcos K Aguilera, Idit Keidar, Dahlia Malkhi, and Alexander Shraer. Dynamic atomic storage without consensus. *Journal of the ACM (JACM)*, 58(2):7, 2011.

2 Eduardo Alchieri, Alysson Bessani, Fabíola Greve, and Joni Fraga. Efficient and modular consensus-free reconfiguration for fault-tolerant storage. *arXiv preprint arXiv:1607.05344*, 2016.

3 Hagit Attiya, Hyun Chul Chung, Faith Ellen, Saptaparni Kumar, and Jennifer L. Welch. Emulating a shared register in a system that never stops changing. *IEEE Trans. Parallel Distrib. Syst.*, 30(3):544–559, 2019.

4 Hagit Attiya, Sweta Kumari, Archit Somani, and Jennifer L. Welch. Store-collect in the presence of continuous churn with application to snapshots and lattice agreement. *CoRR*, abs/2003.07787, 2020. URL: `https://arxiv.org/abs/2003.07787`.

5 Roberto Baldoni, Silvia Bonomi, Anne-Marie Kermarrec, and Michel Raynal. Implementing a register in a dynamic distributed system. In *DISC*, pages 639–647. IEEE, 2009.

6 Roberto Baldoni, Silvia Bonomi, Anne-Marie Kermarrec, and Michel Raynal. Implementing a register in a dynamic distributed system. In *ICDCS*, pages 639–647, 2009.

7 Gabriel Bracha. Asynchronous byzantine agreement protocols. *Information and Computation*, 75(2):130–143, 1987.

8 Gabriel Bracha and Sam Toueg. Asynchronous Consensus and Broadcast Protocols. *JACM*, 32(4), 1985.

9 Christian Cachin, Rachid Guerraoui, and Luís Rodrigues. *Introduction to reliable and secure distributed programming.* Springer Science & Business Media, 2011.

10 Gregory V Chockler, Idit Keidar, and Roman Vitenberg. Group communication specifications: a comprehensive study. *ACM Computing Surveys (CSUR)*, 33(4):427–469, 2001.

11 P. Th. Eugster, R. Guerraoui, S. B. Handurukande, P. Kouznetsov, and A.-M. Kermarrec. Lightweight probabilistic broadcast. *ACM Trans. Comput. Syst.*, 21(4):341–374, November 2003. `doi:10.1145/945506.945507`.

12 Eli Gafni and Dahlia Malkhi. Elastic configuration maintenance via a parsimonious speculating snapshot solution. In *DISC*, pages 140–153. Springer, 2015.

13 Haoyan Geng and Robbert Van Renesse. Sprinkler - Reliable broadcast for geographically dispersed datacenters. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 8275 LNCS:247–266, 2013. `doi:10.1007/978-3-642-45065-5_13`.

14 Rachid Guerraoui, Jovan Komatovic, Petr Kuznetsov, Yvonne-Anne Pignolet, Dragos-Adrian Seredinschi, and Andrei Tonkikh. Dynamic Byzantine Reliable Broadcast [Technical Report]. *arXiv preprint arXiv:2001.06271*, 2020. URL: `https://arxiv.org/abs/2001.06271`.

15 Rachid Guerraoui, Petr Kuznetsov, Matteo Monti, Matej Pavlovič, and Dragos-Adrian Seredinschi. The consensus number of a cryptocurrency. In *PODC*, pages 307–316, 2019.

16 Rachid Guerraoui, Petr Kuznetsov, Matteo Monti, Matej Pavlovic, and Dragos-Adrian Seredinschi. Scalable byzantine reliable broadcast. In *DISC*, pages 22:1–22:16, 2019.

17 Leander Jehl, Roman Vitenberg, and Hein Meling. Smartmerge: A new approach to reconfiguration for atomic storage. In *International Symposium on Distributed Computing*, pages 154–169. Springer, 2015.

18 Anne-Marie Kermarrec and Maarten Van Steen. Gossiping in distributed systems. *ACM SIGOPS operating systems review*, 41(5):2–7, 2007.

19 Saptaparni Kumar and Jennifer L. Welch. Byzantine-tolerant register in a system with continuous churn. *CoRR*, abs/1910.06716, 2019. `arXiv:1910.06716`.

20 Petr Kuznetsov, Thibault Rieutord, and Sara Tucci-Piergiovanni. Reconfigurable lattice agreement and applications. In *OPODIS*, 2019.

21 Petr Kuznetsov and Andrei Tonkikh. Asynchronous reconfiguration with byzantine failures. In *DISC*, pages 27:1–27:17, 2020.

22 Dahlia Malkhi, Michael Merritt, and Ohad Rodeh. Secure Reliable Multicast Protocols in a WAN. In *ICDCS*, 1997.

23 Dahlia Malkhi and Michael Reiter. A high-throughput secure reliable multicast protocol. *Journal of Computer Security*, 5(2):113–127, 1997.

24 Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Whitepaper*, 2008.

25 Fernando Pedone and André Schiper. Handling message semantics with generic broadcast protocols. *Distributed Computing*, 15(2):97–107, 2002.

26 Alexander Spiegelman and Idit Keidar. On liveness of dynamic storage. In *SIROCCO*, pages 356–376. Springer, 2017.

27 Josef Widder and Ulrich Schmid. Booting clock synchronization in partially synchronous systems with hybrid process and link failures. *Distributed Computing*, 20(2):115–140, 2007.