Semi-Passive Replication and Lazy Consensus

Xavier Défago

[*] *School of Information Science, Japan Advanced Institute of Science and Technology (JAIST),*
1-1 Asahidai, Tatsunokuchi, Nomi, Ishikawa 923-1292, Japan,
(phone) +81-76-151 1224 (fax) +81-76-151 1149
[†] *PRESTO, Japan Science and Technology Agency (JST)*

E-mail: defago@jaist.ac.jp

and

André Schiper

Faculté Informatique et Communications, École Polytechnique Fédérale de Lausanne (EPFL),
CH-1015 Lausanne, Switzerland, (phone) +41-21-693 4248 (fax) +41-21-693 6770

E-mail: andre.schiper@epfl.ch

This paper presents two main contributions: semi-passive replication and Lazy Consensus. The former is a replication technique with parsimonious processing. It is based on the latter; a variant of Consensus allowing the lazy evaluation of proposed values.

Semi-passive replication is a replication technique with parsimonious processing. This means that, in the normal case, each request is processed by only one single process. The most significant aspect of semi-passive replication is that it requires a weaker system model than existing techniques of the same family. For semi-passive replication, we give an algorithm based on the Lazy Consensus.

Lazy Consensus is a variant of the Consensus problem that allows the lazy evaluation of proposed values, hence the name. The main difference with Consensus is the introduction of an additional property of laziness. This property requires that proposed values are computed only when they are actually needed. We present an algorithm based on Chandra and Toueg's Consensus algorithm for asynchronous distributed systems with a $\Diamond\mathcal{S}$ failure detector.

Key Words: replication techniques, fault tolerance, high availability, failure detectors, asynchronous systems, consensus, group membership, distributed systems

---

A preliminary version of this paper appeared in *Proc. 17th IEEE Intl. Symp. on Reliable Distributed Systems* (IEEE CS Press, pp. 43–50) [12].

CONTENTS

1. INTRODUCTION

A major problem inherent to distributed systems is their potential vulnerability to failures. Indeed, whenever a single node crashes, the availability of the whole system may be compromised. Interestingly, the distributed nature of those systems also provides the means to *increase* their reliability. Distribution makes it possible to introduce redundancy and, thus, make the overall system more reliable than its individual parts.

Redundancy is usually introduced by the replication of components, or services. Although replication is an intuitive and readily understood concept, its implementation is difficult. Replicating a service in a distributed system requires that each replica of the service keeps a consistent state, which is ensured by a specific replication protocol [21]. There exist two major classes of replication techniques to ensure this consistency: *active* and *passive* replication. Both replication techniques are useful since they have complementary qualities.

With active replication [31], each request is processed by all replicas in the same relative order to ensure that replicas remain consistent. This technique ensures a fast reaction to failures, and sometimes makes it easier to replicate legacy systems. However, active replication uses processing resources heavily and requires the processing of requests to be *deterministic*.[1] This last point is a very strong limitation since, in a program, there exist many potential sources for non-determinism [28]. For instance, multi-threading typically introduces non-determinism.

With passive replication (also called *primary-backup*) [7, 21], only one replica (primary) processes the request, and sends update messages to the other replicas (backups). This technique is important because it uses less resources than active replication does, without the requirement of operation determinism. On the other hand, the replicated service usually has a slow reaction to failures. For instance, when the primary crashes, the failure must be detected by the other replicas, and the request may have to be reprocessed by a new primary. This may result in a significantly higher response time for the request being processed. For this reason, active replication is often considered a better choice for most real-time systems, and passive replication for most other cases [32].

In most computer systems, the implementation of passive replication is based on a synchronous model, or relies on some dedicated hardware device [5, 7, 15, 29, 37]. However, we consider here the context of asynchronous systems in which the detection of failures is not certain. In such systems, all implementations of passive replication that we know of are based on a group membership service and must exclude the primary whenever it is

[1] Determinism means that the result of an operation depends only on the initial state of a replica and the sequence of operations it has already performed.

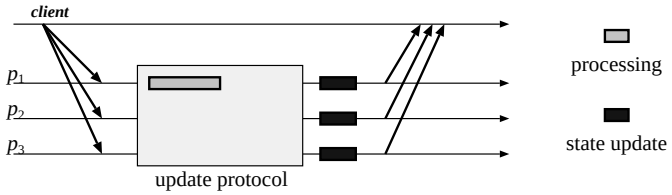

FIG. 1. Semi-passive replication (no crash).
(conceptual representation: the *update protocol* actually hides several messages)

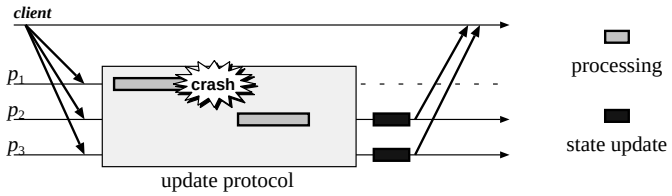

FIG. 2. Semi-passive replication (crash of the coordinator).
(conceptual representation: the *update protocol* actually hides several messages)

suspected to have crashed (e.g., [6, 24, 34]). This is a strong practical limitation of passive replication since this means that a mere suspicion can be turned into a failure, thus reducing the actual fault-tolerance of the system. Conversely, there exist implementations of active replication that neither require a group membership service nor need to kill suspected processes (e.g., active replication based on the Atomic Broadcast algorithm proposed by Chandra and Toueg [8]).

In this paper, we present the semi-passive replication technique; a new technique that retains the essential characteristics of passive replication while avoiding the necessity to force the crash of suspected processes. The most important consequence is that it makes it possible to decouple (1) the replication algorithm from (2) housekeeping issues such as the management of the membership. For instance, this allows the algorithm to use an aggressive failure detection policy in order to react quickly to a crash.

1.1. Overview of Semi-Passive Replication

Semi-passive replication is a variant of passive replication that retains most of its major characteristics (e.g., allows for non-deterministic processing, and requires less processing than active replication). The main difference with passive replication is that the selection of the primary is based on the rotating coordinator paradigm [8] and not on a group membership service as usually done in passive replication. The rotating coordinator mechanism is a simpler mechanism and lower-level mechanism.

Informally, semi-passive replication works as follows. The client sends its request to all replicas p_1, p_2, p_3 (see Fig. 1). The servers know that p_1 is the first primary, so p_1 handles the requests and updates the other servers (the update messages from p_1 to $\{p_2, p_3\}$ are not shown on Fig. 1).

If p_1 crashes and is not able to complete its job as the primary, or if p_1 does not crash but is incorrectly suspected of having crashed, then p_2 takes over as the new primary. The details of how this works are explained later in Section 4. Figure 2 illustrates a scenario in which p_1 crashes after handling the request, but before sending its update message. After the crash of p_1, p_2 becomes the new primary.

These examples do not show which process is the primary for the next client requests, nor what happens if client requests are received concurrently. These issues are explained in detail in Section 4. However, the important point in this solution is that no process is ever excluded from the group of servers (as in a solution based on a membership service). In other words, in case of false suspicion, there is no join (and state transfer) that needs later to be executed by the falsely suspected process. This significantly reduces the cost related to an incorrect failure suspicion, i.e., the cost related to the aggressive timeout option mentioned before.

1.2. Structure of the Paper

The contribution of this paper is twofold: semi-passive replication and Lazy Consensus. For semi-passive replication, we give a definition of the problem and propose an algorithm based on the Lazy Consensus abstraction. Similarly, we define the Lazy Consensus problem, and propose a corresponding algorithm that adapts from the Chandra-Toueg Consensus algorithm for the $\Diamond\mathcal{S}$ failure detector.

The rest of the paper is structured as follows. Section 2 presents the system model considered in this paper, and defines various notations used throughout the paper. Section 3 defines the two problems considered in this paper, namely, semi-passive replication and Lazy Consensus. In Section 4, we present our algorithm for semi-passive replication. In Section 5, we present an algorithm for Lazy Consensus in asynchronous systems augmented with a $\Diamond\mathcal{S}$ failure detector. Section 6 illustrates the execution of our semi-passive replication algorithm with selected scenarios. Section 7 concludes the paper. The two appendices present the correctness proofs of the semi-passive and Lazy Consensus algorithms respectively.

2. SYSTEM MODEL AND DEFINITIONS

In this section, we describe the system model assumed in this paper, and describe important related notations and definitions.

2.1. System Model

We consider a distributed system composed of processes that communicate by exchanging messages only. The system is asynchronous in the sense that there exist bounds neither on communication delays nor on process speed.

We distinguish between two kinds of processes, namely, client processes and server replicas. The set of all clients in the system is denoted by Π_C, and the set of server replicas is denoted by Π_S.[2] The composition of the set Π_S, initially known by all processes, do not change over time although it might include some processes that have crashed. We also denote the number of server processes by $n = |\Pi_S|$. In contrast, there can exist infinitely many client processes in the system.

Processes fail by crashing (i.e., we do not consider Byzantine processes) and crashes are permanent.[3] A correct process is one that does not crash. Processes communicate through quasi-reliable communication channels [3]. Quasi-reliable communication channels guarantee that if a correct process p sends a message m to a correct process q, then q

[2] Note that $\Pi_C \cap \Pi_S$ need not be empty.

[3] In practice, this means that whenever a crashed process recovers from crash, it takes a new identity.

will eventually receive m. In addition, a quasi-reliable channel ensures that messages are (1) not duplicated, (2) not corrupted, and (3) not spuriously created.

Remark. We make these assumptions in order to simplify the description of the algorithms. Indeed, based on the literature, the algorithms can easily be extended to lossy channels and network partitions [3, 1], and to handle process recovery [2, 23, 25]. However, this would obscure the key idea of semi-passive replication by introducing unnecessary complexity.

2.2. Failure Detectors

Formally, it is impossible for processes to reach agreement (i.e., solve Consensus) deterministically in an asynchronous distributed system where some processes can crash [18]. This impossibility stems from the fact that, in such a system, a crashed process cannot be distinguished from a very slow one. It follows that, the ability to detect the crash of processes is a fundamental issue.

In this section, we present three related approaches to detect the crash of processes in a distributed system. We begin with unreliable failure detectors as this is the basis for the algorithms presented in this paper.

2.2.1. Unreliable Failure Detectors

The impossibility result mentioned above also applies to Lazy Consensus. Hence, in order to solve Lazy Consensus among the server processes, we consider that the system is augmented with some unreliable failure detector [8] that runs between the processes in Π_S. In particular, we assume a failure detector of class $\Diamond\mathcal{S}$, sufficient to solve the Consensus problem, and defined over Π_S by the following properties [8]:

(STRONG COMPLETENESS) There is a time after which every process in Π_S that crashes is permanently suspected by all correct processes in Π_S.

(EVENTUAL WEAK ACCURACY) There is a time after which some correct process in Π_S is never suspected by any correct process in Π_S.

2.2.2. Perfect Failure Detectors

Many replication algorithms rely on the ability to detect process failures accurately. More specifically, they rely on the availability of a perfect failure detector. In contrast with unreliable failure detectors, a perfect failure detector is one whereby no process suspects a process that has not crashed. A failure detector of class $\mathcal{P}$ (i.e., a perfect failure detector) must enforce the property of strong completeness described above, and the following property of strong accuracy [8]:

(STRONG ACCURACY) No process is suspected before it has crashed.

In practice, a perfect failure detector can be emulated in an asynchronous system by relying on timeouts and the ability to control, in particular provoke, the crash of processes [17]. However, although technically possible, this is also mostly undesirable, as this potentially degrades the overall stability of the system (see [14] for details).

2.2.3. Group Membership

A group membership is a service that usually combines two different purposes (see [10] for a detailed survey). On the one hand, a group membership is used to allow processes to

join and leave the computation dynamically. On the other hand, group membership is used as a way to detect the crash of processes. The main difference with failure detectors is that, unlike the latter, a group membership provides *consistent* information about failures. This often requires to exclude suspected processes from the group and consider as crashed and ask them to take a new identity. A group membership is often used as a way to emulate a perfect failure detector.

Essentially, providing consistent information about failures places group membership at a higher level of abstraction than failure detectors. This difference in structure leads to difference in behavior. A recent study by Urbán et al. [33] compares the two models (i.e., group membership and failure detectors) using Total Order Broadcast[4] as a reference. Among other things, the study shows that, unlike a common belief, the overall performance in failure-free runs of Total Order Broadcast do not change whether it is based on group membership (optimized fixed sequencer algorithm) or unreliable failure detectors (optimized Consensus-based destinations agreement algorithm). However, the study shows that the solution based on unreliable failure detectors is several orders of magnitude more robust to wrong suspicions. In particular, this means that more aggressive failure detectors can be used, thus resulting in far better failover time in the occurrence of failures.

2.3. Replication Model

Without loss of generality, we define replication in the client-server model. We consider a model in which each process is modeled as a state machine. There are two types of processes: clients and server replicas. Clients execute the following two external events:

- $send(req)$, the emission of the request req by a client; and
- $receive(resp_{req})$, the reception by a client of the response to request req (message $resp_{req}$).

Server replicas execute the following two events:

- $handle(req)$, the processing of request req that generates an *update message* upd_{req};
- $update(req)$, the modification of the state of the replica as the result of processing req.

This must be deterministic.

We also introduce important notations to describe the replicated server. This notation is used to express the semi-passive replication algorithm in Section 4.

- req: request message sent by a client (denoted by $sender(req)$).
- upd_{req}: update message generated by a server after handling request req.
- $resp_{req}$: response message to the client $sender(req)$, generated by a server after handling request req.
- $state_s$: the state of the server process s.
- $handle : (req, state_s) \longmapsto (upd_{req}, resp_{req})$: Processing of request req by the server s in $state_s$. The result is an update message upd_{req} and the corresponding response message $resp_{req}$.

[4] Total Order Broadcast, also known as Atomic Broadcast, is an agreement problem at the core of active replication. Roughly speaking, messages are broadcasted concurrently, and all destination processes must deliver the same set of message in the same relative order. A broad survey [13] has been written on the topic.

• $update : (upd_{req}, state_{s'}) \longmapsto state'_{s'}$: Returns a new state $state'_{s'}$, obtained by the application of the update message upd_{req} to the state $state_{s'}$. This corresponds to the event $update(req)$ mentioned above, where s' is the server that executes *update*.

2.4. Sequences

The algorithms presented in this paper rely on sequences. A sequence is a finite ordered list of elements. With a few minor exceptions, the notation defined here is borrowed from that of Gries and Schneider [20].

A sequence of three elements a, b, c is denoted by the tuple $\langle a, b, c\rangle$. The symbol ϵ denotes the empty sequence. The length of a sequence seq is the number of elements in seq and is denoted $\#seq$. For instance, $\#\langle a, b, c\rangle = 3$, and $\#\epsilon = 0$.

Elements can be added either at the beginning or at the end of a sequence. Adding an element e at the beginning of a sequence seq is called prepending (see [20]) and is denoted by $e \lhd seq$. Similarly, adding an element e at the end of a sequence seq is called appending and is denoted by $seq \rhd e$.

We define the operator $[\,]$ for accessing a single element of the sequence. Given a sequence seq, $seq[i]$ returns the i^{th} element of seq. The element $seq[1]$ is then the first element of the sequence, and is also denoted as $head.seq$. The tail of a non-empty sequence seq is the sequence that results from removing the first element of seq. Thus, we have

$$seq = head.seq \lhd tail.seq$$

For convenience, we also define the following additional operations on sequences. First, given an element e and a sequence seq, the element e is a member of seq (denoted $e \in seq$) if e is a member of the set composed of all elements of seq. Second, given a sequence seq and a set of elements S, the exclusion $seq - S$ is the sequence that results from removing from seq all elements that appear in S.

3. PROBLEM SPECIFICATIONS

This section presents the specification of the two problems addressed in this paper. First, we present the specification of semi-passive replication. Second, we present the problem of Lazy Consensus.

3.1. Specification of Semi-Passive Replication

The definition below is based on a specification framework for replication techniques described by Défago [14],[5] of which we only present the relevant parts here.

3.1.1. Generic Replication Problem

First of all, replications techniques are defined by the Generic Replication Problem. This part of the specification is common to replication techniques, regardless of their strategies (e.g., active replication, passive replication). The specificity of a given strategy is captured by extending the definition with additional properties.

[5] The definition of the total order property was in fact adapted from a property called "gap-free uniform total order" proposed by Aguilera et al. [4] for the problem of Total Order Broadcast.

(TERMINATION) If a correct client $c \in \Pi_C$ sends a request, it eventually receives a reply.

(TOTAL ORDER) For any two requests req and req', if some replica executes $update(req')$ after $update(req)$, then a replica executes $update(req')$ only after it has executed $update(req)$.

(UPDATE INTEGRITY) For any request req, every replica executes $update(req)$ at most once, and only if $send(req)$ was previously executed by a client.

(RESPONSE INTEGRITY) For any event $receive(resp_{req})$ executed by a client, the event $update(req)$ is executed by some correct replica.

A given replication technique will operate correctly as long as it satisfies the four properties above.

3.1.2. Passive and Semi-Passive Replication

As already mentioned, the specification above is common to replication techniques, regardless of their approach. Hence, the specificity of a given strategy is captured by extending the specification with additional properties. We define both passive and semi-passive replication with an additional property of *parsimony*.

Passive replication, as for instance described by Budhiraja et al. [7], is expressed in a model with perfect failure detection. In particular, they require that no more than one server replica can be the primary at any time. This is expressed by the following property of parsimony.

(STRONG PARSIMONY) If a request req is processed by a replica p, then no other replica processes req unless p crashes.

Enforcing strong parsimony requires a way to detect, with absolute certainty, the crash of other processes. In other words, strong parsimony requires a perfect failure detector (see Sect. 2.2.2).

In contrast, semi-passive replication is defined with a weaker property that relates parsimony to the *detection* of failures rather than their *occurrence*. The definition is expressed as follows.

(WEAK PARSIMONY) If the same request req is processed by two replicas p and q, then at least one of p and q is suspected by some replica.

It follows that the parsimony of a semi-passive replication algorithm is related to the failure detection provided by the system model. In particular, it is easy to see that, under a perfect failure detector, weak and strong parsimony are in fact identical.

3.2. Specification of Lazy Consensus

The Lazy Consensus problem is a generalization over the Consensus problem that allows processes to delay the computation of their initial value. In the traditional definition of Consensus (e.g., [18, 8]), a process begins the problem with an initial value. In contrast, with the definition of Lazy Consensus, a process begins without initial value. The initial value of the process is computed only when it becomes necessary, if at all.[6]

[6] The problem is called "Lazy Consensus" in reference to its similarities with the programming technique known as "lazy evaluation."

The Lazy Consensus problem is defined here as a problem among server processes, that is, we consider only the set of processes Π_S. Processes propose no value initially, but instead provide the algorithm with an argument-less function that computes and returns a proposed value when called. More concretely, processes begin the problem by calling the procedure *LazyConsensus*(giv), where giv is an argument-less function[7] that, when called, computes an initial value v (with $v \neq \perp$[8]) and returns it. When the algorithm calls giv on behalf of process p, we say that p *proposes* the value v returned by giv. When a process q executes $decide(v)$, we say that q *decides* the value v. The Lazy Consensus problem is specified in Π_S by the following properties:

(TERMINATION) Every correct process eventually decides some value.
(UNIFORM INTEGRITY) Every process decides at most once.
(AGREEMENT) No two correct processes decide differently.
(UNIFORM VALIDITY) If a process decides v, then v was proposed by some process.
(PROPOSITION INTEGRITY) Every process proposes a value at most once.
(WEAK LAZINESS) If two processes p and q propose a value, then at least one of p and q is suspected by some[9] process in Π_S.

Laziness is the only new property with respect to the standard definition of the Consensus problem [8]. In Section 4, we present an algorithm for semi-passive replication that uses Lazy Consensus. Solving Lazy Consensus is discussed in Section 5.

Remark. Alternatively, stronger definitions of Lazy Consensus problems can be given, by requiring stronger definitions of laziness. Thus, we define the *quasi-strong Lazy consensus* and the *strong Lazy consensus* as Lazy consensus problems that respectively satisfy the following laziness properties:

(QUASI-STRONG LAZINESS) If two processes p and q propose a value, then p and q are not both correct.
(STRONG LAZINESS) If a process p proposes a value, then no process q proposes a value before p has crashed unless q has crashed before p proposes a value.

4. SEMI-PASSIVE REPLICATION ALGORITHM

We begin this section by giving a general overview of the semi-passive replication algorithm. We then present our algorithm for semi-passive replication, expressed as a sequence of Lazy consensus problems. Finally, we prove and discuss the parsimony property of the semi-passive replication algorithm (the correctness of the algorithm is proved in the appendix).

4.1. Basic Idea: Consensus on "update" values

As mentioned in Section 1.1, in the semi-passive replication technique, the requests are handled by a single process; the primary. After the processing of each request, the primary sends an *update* message to the backups, as illustrated on Figure 3.

[7] giv stands for *get initial value*.
[8] The symbol $\perp$ (bottom) is a common way to denote the absence of value. This is called either *nil* or *null* in most programming languages.
[9] As a matter of fact, the Lazy Consensus algorithm presented in this paper satisfies a stronger property: two processes propose a value only if one of them is suspected by a *majority* of processes in Π_S (Lemma 2.11, p. 27).

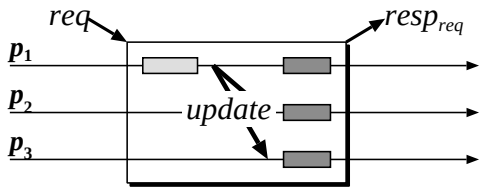

FIG. 3. Semi-passive replication: update message sent by the primary.

Our solution is based on a sequence of Lazy Consensus problems, in which every instance decides on the *content of the update message*. This means that the initial value of every consensus problem is an *update value*, generated when handling the request. The cost related to getting the initial value is high as it requires the processing of the request. So, we want to avoid a situation in which each server processes the request, i.e., has an initial value for consensus (or else the semi-passive replication technique could no more be qualified as "parsimonious"). This explains the need for a "laziness" property regarding the Consensus problem.

Expressing semi-passive replication as a sequence of Lazy Consensus problems hides inside the consensus algorithm the issue of selecting a primary. A process p takes the role of the primary (i.e., handles client requests) exactly when it proposes its initial value for Consensus.

4.2. Semi-Passive Replication Algorithm

The algorithm for semi-passive replication relies on the laziness property of the Lazy Consensus. The laziness property of Lazy Consensus is the key to satisfy parsimonious processing (see Sect. 4.3, p. 11). However, laziness does not affect the correctness of the algorithm as a *Generic Replication* problem (see Sect. A.1, p. 22; Remark 4.3, p. 12)

Variables. Every server s manages an integer k (line 5), which identifies the current instance of the Lazy Consensus problem. Every server process also handles the variables $recvQ$ and $hand$ (lines 2,3):

- $recvQ_s$ is a sequence (receive queue) containing the requests received by a server s, from the clients.
- $hand_s$ is a set which consists of the requests that have been processed.

Algorithm description. We now give a textual description of the algorithm. The pseudo-code is expressed in Algorithm 1. Briefly speaking, the algorithm relies on a sequence of Lazy Consensus executions and works as follows:

- When a server s receives a new request req from a client, that request is simply appended to the receive queue $recvQ_s$ of that server, unless it was previously received and/or handled.
- Whenever the receive queue $recvQ_s$ is not empty and the last execution of the Lazy Consensus has finished, a new instance of the Lazy Consensus is started. The proposition function *handleRequest()*, invoked lazily by the Lazy Consensus algorithm, takes the first request req from the receive queue, handles it, and returns a tuple $(req, upd_{req}, resp_{req})$ containing the request req, an update message upd_{req}, and a reply $resp_{req}$ for the client. The decision value of the Lazy Consensus is one such tuple.

• When a server s receives the decision value $(req, upd_{req}, resp_{req})$ of an execution of the Lazy Consensus, it forwards the reply message $resp_{req}$ to the client, updates its state according to the update message upd_{req}, and moves the request req from the receive queue $recvQ_s$ to the set of handled requests $hand_s$.

ALGORITHM 1 (SEMI-PASSIVE REPLICATION (CODE OF SERVER s)).

```
1.  Initialization:
2.    recvQ_s ← ϵ                          {sequence of received requests, initially empty}
3.    hand_s ← ∅                           {set of handled requests}
4.    state_s ← state^0
5.    k ← 0

6.  function handleRequest()
7.    req ← head.recvQ_s
8.    (upd_req, resp_req) ← handle(req, state_s)
9.    return (req, upd_req, resp_req)
10. end handleRequest()

11. when receive(req_c) from client c                                   /TASK 1/
12.   if req_c ∉ hand_s ∧ req_c ∉ recvQ_s then
13.     recvQ_s ← recvQ_s ▷ req_c
14.   end if
15. end when

16. when #recvQ_s > 0                                                   /TASK 2/
17.   k ← k + 1
18.   LazyConsensus(k,handleRequest)       {Solve the k^th Lazy consensus}
19.   wait until decide(k, (req, upd_req, resp_req))
20.   send (resp_req) to sender(req)       {Send response to client}
21.   state_s ← update(upd_req, state_s)   {Update the state}
22.   recvQ_s ← recvQ_s − {req}
23.   hand_s ← hand_s ∪ {req}
24. end when
```

4.3. Parsimony of the Semi-Passive Replication Algorithm

As mentioned earlier, the semi-passive replication algorithm only relies on the laziness of the Lazy Consensus in order to satisfy the Parsimony property of semi-passive replication. This means that laziness is the key to parsimonious processing, but it does not influence the safety properties of the algorithm. In other words, even if the algorithm relies on a Consensus algorithm which does not satisfy any laziness property, the replication algorithm still satisfies the properties of the generic replication problem discussed in Section 3.1 (but it might not satisfy the *parsimonious processing* property, Sect. 3.1.2).

THEOREM 1.1. Algorithm 1 solves the generic replication problem (defined in Section 3.1).

The details of the proof are given in the appendix (pp. 22–24). It is nevertheless important to note that Theorem 1.1 is proved independently of the laziness property of the Lazy Consensus.

LEMMA 4.1. *Algorithm 1 with* weak *Lazy Consensus satisfies* weak *parsimony.*

Proof. Processes process a request at line 8, i.e., when they propose a value. Therefore, the *weak parsimony* property follows directly from the *weak laziness* property of the Lazy Consensus. ■

THEOREM 4.1. *Algorithm 1 with* weak *Lazy Consensus solves the* semi-passive replication *problem.*

Proof. Follows directly from Theorem 1.1 (generic replication) and Lemma 4.1 (weak parsimony). ■

We now show that implementing passive replication based on Algorithm 1 merely consists in relying on a strong Lazy Consensus algorithm (see Sect. 3.2).

LEMMA 4.2. *Algorithm 1 with* strong *Lazy Consensus satisfies* strong *parsimony.*

Proof. The proof is a trivial adaptation from that of Lemma 4.1. ■

COROLLARY 4.1. *Algorithm 1 with* strong *Lazy Consensus solves the* passive replication *problem.*

Proof. Follows directly from Theorem 1.1 (generic replication) and Lemma 4.2 (strong parsimony). ■

Remark. An interesting (and potentially controversial) point to raise here is that the property of parsimony in itself is merely a question of quality of service rather than actual correctness. Indeed, as long as the server solves the Generic Replication problem, it will continue to operate devoid of any inconsistencies even if laziness is not satisfied.

If not for our algorithm, this remark would be quite pointless since other passive replication algorithms cannot separate both issues (generic replication and parsimony). In contrast, our algorithm presents these issues as being orthogonal.

5. SOLVING LAZY CONSENSUS

In this section, we give an algorithm that solves the problem of Lazy Consensus defined in Section 3.2.[10] The algorithm presented here is adapted from the Chandra-Toueg consensus algorithm for $\Diamond\mathcal{S}$ [8]. Both algorithms rely on the assumption that at least a majority of the participating processes are correct.

To better describe the difference between the Chandra-Toueg algorithm and ours, we begin the section with an informal description of the former algorithm, followed by an equally informal description of the algorithm for Lazy consensus.

Then, we describe two simple yet important optimizations that can be applied to both algorithms. The first optimization reduces the first round by one phase, whereas the second optimization improves the selection of coordinators when several instances of the consensus algorithm are executed in sequence.

Finally, we describe the complete pseudo-code for our Lazy consensus algorithm, which incorporates the two optimizations mentioned above. The adapted proofs of correctness are presented in Appendix A.2.

5.1. Chandra-Toueg Consensus Algorithm using $\Diamond\mathcal{S}$

The Chandra-Toueg [8] consensus algorithm described here assumes a failure detector of class $\Diamond\mathcal{S}$ and that no less than a majority of the processes in Π_S are correct. Figure 4

[10] An earlier version of this algorithm was called $\mathcal{DIV}$consensus [12]. Note that $\mathcal{DIV}$consensus used to designate an *algorithm*, whereas Lazy Consensus now designates a *problem*.

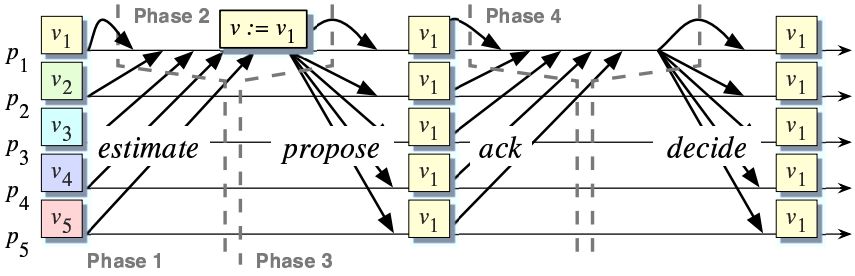

FIG. 4. Chandra-Toueg Consensus; illustration of a single round execution.

presents the communication generated by the algorithm in a failure/suspicion-free run. The figure depicts the four phases that constitute the first round of the protocol. The algorithm is now described informally.

The algorithm proceeds through a sequence of asynchronous rounds. Each round is uniquely identified by a sequence number, and all protocol messages are identified by the number of the round to which they belong. Being asynchronous, several rounds can actually take place simultaneously, although they are logically ordered by their sequence number. In each round one of the processes in Π_S is defined as a coordinator for that round. The composition of Π_S never changes and is assumed to be initially known to all processes. Hence, the coordinator of round r is designated deterministically by the formula[11] $c^r = ((r-1) \bmod n) + 1$, thus cycling among the set of processes. This is commonly known as the rotating coordinator paradigm.

Processes begin the execution of the consensus with the $propose$ event and some proposition value v_0. Each process maintains several variables, the most important of which are: (1) the number of the current round, (2) an estimate of the decision value, and (3) a logical timestamp associated with the estimate. The processes begin the first round of the algorithm with the variables set to 1, v_0, and 0, respectively.

- In Phase 1, all processes in Π_S send their estimate to the coordinator of the current round, timestamped with the round number in which they last modified it.
- In Phase 2, the coordinator waits for a proposition from a majority of the processes in Π_S. It selects the estimate with the highest timestamp and modifies its own estimate accordingly (breaking ties can be done arbitrarily). The coordinator then broadcasts its estimate as its proposition for the decision value.
- In phase 3, the processes wait for a proposition from the coordinator. They adopt the value proposed by the coordinator by changing their estimate and using the round number as the new timestamp. Then, they acknowledge the proposition and proceed to the first phase of the next round.

In case a process suspects the coordinator before it receives a proposition, that process sends a *negative* acknowledgment before proceeding to the first phase of the next round.

- In Phase 4, the coordinator waits until it has received an acknowledgment message (positive or negative) from a majority of the processes. If all received acknowledgments

[11] To be exact, Chandra and Toueg [8] use the slightly simpler formula $c^r = (r \bmod n) + 1$, which counter-intuitively designates p_2 as the coordinator of round 1, p_3 for round 2, and so forth.

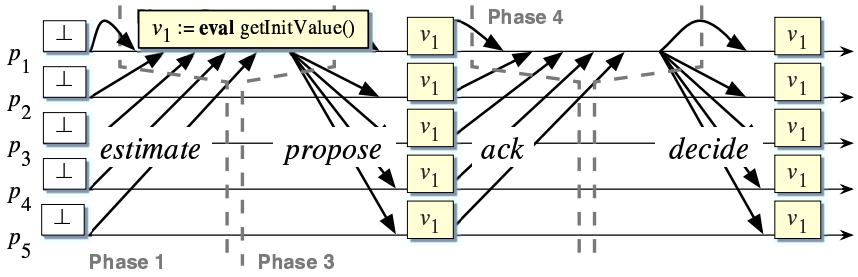

FIG. 5. Lazy Consensus; illustration of a single round execution. Initially, the processes hold $\perp$ instead of a proposition value.
In the first round, estimate messages of the first phase are not essential to the algorithm (discussed in Sect. 5.3.1).

are positive, the proposed value becomes the decision value. The coordinator then informs the other processes by broadcasting the decision value using Reliable Broadcast.

In contrast, if one of the received acknowledgments is negative, the coordinator gives up and proceeds directly to the first phase of the next round.

5.2. Lazy Consensus Algorithm (informal description)

The Lazy consensus algorithm described in this paper is an adaptation of the Chandra-Toueg algorithm that shares the same assumptions. Rather than describing the whole algorithm, we simply present the most significant differences. Figure 5 presents the first round of the protocol in a failure/suspicion-free run. Notice that, for the sake of simplicity, this section presents a simplified version of the algorithm.

In the Lazy consensus algorithm, processes begin the execution of the algorithm by proposing a function (or a lambda closure) called giv which, if called, computes a proposition value and returns it. Other than that, processes maintain the same variables as in the Chandra-Toueg algorithm, namely, (1) the number of the current round, (2) an estimate of the decision value, and (3) a logical timestamp associated with the estimate. Unlike Chandra-Toueg, processes do not begin with a proposition value, and hence set their estimate to $\perp$, thus representing the absence of a value.

The rest of the algorithm is the same as with Chandra-Toueg's, with the following exception. In Phase 2, the coordinator of the round gathers estimate messages from a majority of processes. Among the estimates received and including its own, the coordinator takes the one, different from $\perp$, that has the highest timestamp. If no such estimate exists, because they are all equal to $\perp$, then the coordinator computes its proposition value by calling the function giv. It then sets its own estimate to the return value of the function and uses that value as its proposition for the round.

Doing so ensures that the function giv is called only when necessary. In fact, it is not difficult to see that any single process will call the function at most once. Beside, in the worst case, the function can only be called by about half of the processes plus one. Intuitively, this is because, if a majority of the processes have called that function, then the coordinator of any subsequent round will receive at least one estimate different from $\perp$ in the second phase of their round.

5.3. Optimizations

The full algorithm (presented in Section 5.4) includes two important optimizations that we present now. The first optimization reduces the overhead of the protocol in failure-free

runs. The second optimization is concerned with situations where several executions of the algorithm are performed in sequence, and the performance penalty that is associated with the crash of the first processes.

5.3.1. Optimization of the First Phase

As observed by Schiper [30], the first phase in the first round of the Chandra-Toueg consensus algorithm (see Sect. 5.1) is not essential for the algorithm. The reason is that, in the first phase, it is known by all processes that the estimate of every processes is their proposition value, timestamped with zero. Hence, when the coordinator collects the estimates in phase two, it can pick any of the estimates as the proposition value. In particular, the coordinator can select *its own* estimate as the proposition value, regardless of the estimates sent by other processes.

Similarly, with Lazy Consensus, all processes start with the value $\perp$ as their estimate. Consequently, the coordinator of the first phase cannot expect anything but $\perp$ from the other processes. Hence, in the first round, the algorithm skips the first phase and proceeds directly to the second phase.

Notice that this optimization applies only to the first round. It is nevertheless useful as, during a failure-free and suspicion-free execution, the latency degree of the protocol is determined by the first round only.

5.3.2. Adaptive rotating coordinator

Several important algorithms involve a sequence of consensus executions. In addition to the semi-passive replication algorithm described in this paper, this is also the case with several Total Order Broadcast algorithms (e.g., [8, 19]), Generic Broadcast [26], some consensus-based group membership services [10], fault-tolerant mobile agents [27].

Unfortunately, in this situation, there is a practical problem inherent to the use of the rotating coordinator. In the rotating coordinator paradigm, every instance of the consensus algorithm selects a coordinator by cycling through processes always in the same sequence, say $\langle p_1, \ldots, p_n \rangle$. This means that p_1 is coordinator for round 1, p_2 for round 2, etc. Assume now that p_1 crashes before consensus number k, then consensus k and *every further* execution of the consensus will always fail in the first round (p_1, the coordinator of round 1 has crashed), hence always requiring at least two rounds to decide. This extra cost (two rounds instead of one) cannot be easily avoided for consensus number k. However, the cost can be avoided for consensus $k+1$ and after, by a simple modification to the rotating coordinator that incurs no additional message.

Let us illustrate this with an example. Consider that, for consensus number k, the processes of Π_S are ordered as follows: $\langle p_1, p_2, p_3, p_4, p_5 \rangle$, which defines p_1 as the first coordinator (see Fig. 6). Assume that p_1 crashes just before the execution of consensus k, and thus the first round fails. Assume again that, after consensus k, all processes can agree on the following permutation of the processes in Π_S: $\langle p_2, p_3, p_4, p_5, p_1 \rangle$. Then, if consensus $k+1$ uses the new permutation, then p_2 becomes the coordinator of the first round and consensus $k+1$ can be solved in one single round in spite of the crash of p_1.

Obviously, reaching an agreement on a new permutation for the rotating sequence requires exactly this,... reaching an agreement. The idea of our optimization is that, during consensus k, processes reach an agreement not only on the decision value for consensus k, but also on a permutation vector to be used during the *next execution* of the consensus, that is, consensus number $k+1$. In fact, the permutation vector can be seen as an implicit part

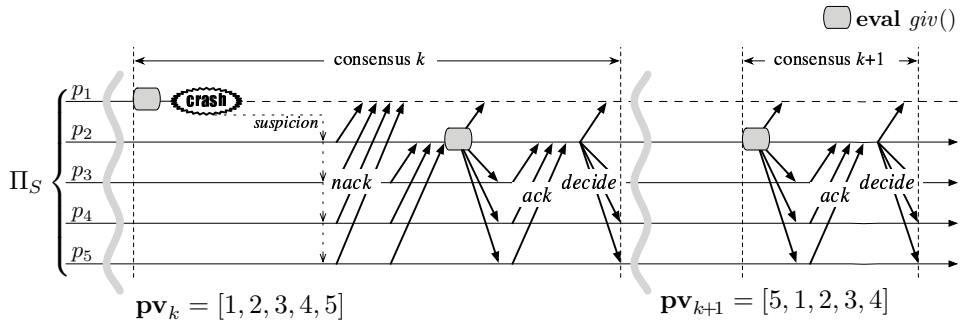

FIG. 6. Permutations of Π_S and selection of a coordinator.

of the decision value. As a result, the agreement on the permutation generates *no additional message*.

More concretely, this occurs as follows. The processes start consensus k with a permutation vector $\mathbf{pv}^k$ agreed by all processes. For the first execution of the consensus, the permutation vector $\mathbf{pv}^1$ is determined statically as being the identity $[1, 2, \ldots, n]$. Then, each execution k of the consensus agrees on the permutation vector for the next execution $\mathbf{pv}^{k+1}$. A permutation vector $\mathbf{pv}^k$ is used during the execution of consensus k to determine the coordinator of round r as $c^r = \mathbf{pv}^k\,[((r-1) \bmod n) + 1]$. During consensus k, the agreement on the next permutation vector $\mathbf{pv}^{k+1}$ occurs as follows. The processes manage two estimate variables instead of a single one: *estV*$_p$ for the decision value, and *estP*$_p$ for the permutation vector. When a coordinator proposes a value, it also proposes a permutation vector with itself as the first coordinator (this is done in Alg. 2 at lines 6 and 32). When the consensus decides, the agreed permutation vector becomes $\mathbf{pv}^{k+1}$ and is used later, for the execution of consensus $k + 1$.

Because a crashed process p cannot propose a value after it has crashed, it is easy to see that p does not remain the first coordinator for more than one entire consensus execution after it has crashed.

Remark. One could possibly mistake the adaptive rotating coordinator for a form of group membership. To prevent this misconception, we would like to emphasize here that adaptive rotating coordinator is merely an extension to the rotating coordinator paradigm and by no means a replacement for a group membership. The latter is indeed a higher-level abstraction, and hence differs by several fundamental aspects.

First and most importantly, with the adaptive rotating coordinator, the composition of the set of processes is static and hence never changes. This is clearly unlike group membership whose primary role is to allow the dynamic join and leave of processes during the computation.

Second, specifications of group membership [10] include the notion of view synchrony that imposes some restrictions on the delivery of application messages. In contrast, this notion is irrelevant to the adaptive rotating coordinator.

Third, a secondary role of a group membership service is to ensure that system resources (i.e., retransmission buffer emptied, etc) are eventually reclaimed. Again, the adaptive rotating coordinator has nothing to do with resource management as this occurs at a different abstraction level.

Finally, with group membership, the agreement on the composition of the group can occur independently from the execution of group communication protocols. In contrast, the mechanism of the adaptive rotating coordinator is embedded within the consensus protocol and cannot occur independently.

Remark. Note that we have presented the idea of the adaptive rotating coordinator using a simple reordering policy. This is enough to illustrate the idea but it is possible, in practice, to use better strategies for the permutation. Changing the reordering policy does not compromise the correctness of the algorithms, as long as the permutation vector is modified only at line 6 and 32 in Algorithm 2.

5.4. Lazy Consensus Algorithm with $\Diamond\mathcal{S}$

We now describe the complete algorithm in more details. Algorithm 2 (page 18) solves the Lazy Consensus problem with a $\Diamond\mathcal{S}$ failure detector and the assumption that at least a majority of the processes in Π_S are correct.

5.4.1. Variables

We first present variables that are retained between execution instances of the algorithm. These variables are global within a single process, but not shared among processes.

- $\mathbf{pv}^k$ represents the permutation vector for consensus instance k. It is determined during consensus execution $k-1$.
- $\mathbf{pv}^1$ is set initially by all processes to be the identity vector, that is, $[1, 2, \ldots, n]$. It is used as the permutation vector for the first consensus execution, that is, instance 1.

The consensus is initiated by calling the procedure *LazyConsensus*, which takes two arguments. The first argument is the instance number k. The second argument is an argument-less function, or closure, called giv. When evaluated, giv computes and returns a proposition value $v \neq \perp$ (see Sect. 3.2). When a process p executes $v_p \leftarrow \mathbf{eval}\ giv$, we say that the process proposes the value v_p.

The following variables are local to procedure *LazyConsensus* and play an important role in the algorithm.

- *estV*$_p$ is the estimate that process p has about the decision value.
- *estP*$_p$ is the estimate that process p has about the next permutation vector.
- r_p is the round number, initially set to 0, but incremented before beginning the round.
- ts_p is the round number when the estimates $(\textit{estV}_p, \textit{estP}_p)$ were last changed. It is initially set to 0.

5.4.2. Algorithm description

We now give a brief description of each phase of the algorithm. Notice that phases 3 and 4 are nearly unchanged from the Chandra-Toueg algorithm described in Section 5.1.

- In Phase 1, all processes in Π_S send their estimates *estV*$_p$ and *estP*$_p$ to the coordinator of the current round, timestamped with the round number in which they last modified them. According to the optimization of Sect. 5.3.2, the first phase is entirely skipped during the first round.

• In Phase 2, the coordinator waits for a proposition from a majority of the processes in Π_S, except during the first round when the coordinator has nothing to wait for (optimization of Sect. 5.3.2). In the receive statement, $\underline{k}$ and $\underline{r_p}$ are pattern matching arguments, i.e., the process waits for a message with the given k and r_p value. The other arguments are output arguments. The coordinator filters the received estimates $estV_q$ and its own. If at least one of them is defined ($\neq \perp$), then the coordinator selects the estimates $(estV_q, estP_q)$ with the highest timestamp and modifies its own estimates $(estV_p, estP_p)$ accordingly. Conversely, if all of the estimates received in the phase are undefined ($= \perp$), then the coordinator proposes a value by evaluating the function giv, and sets its estimate $estV_p$ to the return value of the function. After that, the coordinator broadcasts its estimates $(estV_p, estP_p)$.

• In phase 3, the processes wait for a proposition from the coordinator. They adopt the value proposed by the coordinator by changing their estimates $(estV_{c_p}, estP_{c_p})$, using the round number as the new timestamp. Then, they acknowledge the proposition and proceed to the first phase of the next round. In case a process suspects the coordinator before it receives a proposition, that process sends a *negative* acknowledgment before it proceeds to the first phase of the next round.

• In Phase 4, the coordinator waits for an acknowledgment from a majority of the processes. If all received acknowledgments are positive, the proposed value $(estV_p, estP_p)$ becomes the decision value and the coordinator informs the other processes by broadcasting the decision value using Reliable Broadcast. On the other hand, if one of the received acknowledgments is negative, no decision is taken and the coordinator proceeds directly to the first phase of the next round.

• Upon receiving the decision message with $(estV_q, estP_q)$, a process decides $estV_q$ and sets the permutation vector $\mathbf{pv}^{k+1}$ to $estP_q$. The permutation vector $\mathbf{pv}^{k+1}$ is used for the next consensus execution $k+1$.

ALGORITHM 2 (LAZY CONSENSUS (CODE OF PROCESS p).).

```
1.  Initialization:
2.    pv^1 ← [1, 2, ..., n]

3.  procedure LazyConsensus (k, function giv : ∅ ↦ v)          {code for consensus instance k}
4.    pv^k := permutation vector obtained during instance k − 1
5.    estV_p ← ⊥                                               {p's estimate of the decision value}
6.    estP_p ← {rotate pv^k until p is first}
7.    state_p ← undecided
8.    r_p ← 0                                                  {r_p is p's current round number}
9.    ts_p ← 0                          {ts_p is the last round in which p updated (estV_p, estP_p), initially 0}

10.   while state_p = undecided do                  {rotate through coordinators until decision reached}
11.     c_p ← pv^k [(r_p mod n) + 1]                                {c_p is the current coordinator}
12.     r_p ← r_p + 1

13.     Phase 1:                                {all processes p send (estV_p, estP_p) to the current coordinator}
14.       if r_p > 1 then
15.         send (k, p, r_p, estV_p, estP_p, ts_p) to c_p
16.       end if

17.     Phase 2:                {coordinator gathers ⌈(n+1)/2⌉ estimates and proposes new estimate}
18.       if p = c_p then
19.         if r_p = 1 then
20.           estV_p ← eval giv()                                {p proposes a value}
```

21. **else**
22. **wait until** [for $\left\lceil \frac{(n+1)}{2} \right\rceil$ processes q : received $(\underline{k}, q, \underline{r_p}, \mathit{estV}_q, \mathit{estP}_q, ts_q)$ from q]
23. $msgs_p[r_p] \leftarrow \{(k, q, r_p, \mathit{estV}_q, \mathit{estP}_q, ts_q) \mid p$ received $(k, q, r_p, \mathit{estV}_q, \mathit{estP}_q, ts_q)$ from $q\}$
24. $t \leftarrow$ largest ts_q such that $(k, q, r_p, \mathit{estV}_q, \mathit{estP}_q, ts_q) \in msgs_p[r_p]$
25. **if** $\mathit{estV}_p = \perp$ **and** $\forall (k, q, r_p, \mathit{estV}_q, \mathit{estP}_q, ts_q) \in msgs_p[r_p] : \mathit{estV}_q = \perp$ **then**
26. $\mathit{estV}_p \leftarrow$ **eval** $giv()$ {p proposes *a value*}
27. **else**
28. $\mathit{estV}_p \leftarrow$ select one $\mathit{estV}_q \neq \perp$ s.t. $(k, q, r_p, \mathit{estV}_q, \mathit{estP}_q, t) \in msgs_p[r_p]$
29. $\mathit{estP}_p \leftarrow \mathit{estP}_q$
30. **end if**
31. **end if**
32. send $(k, p, r_p, \mathit{estV}_p, \mathit{estP}_p)$ to all
33. **end if**

34. **Phase 3:** {*all processes wait for new estimate proposed by current coordinator*}
35. **wait until** [received $(\underline{k}, \underline{c_p}, \underline{r_p}, \mathit{estV}_{c_p}, \mathit{estP}_{c_p})$ from c_p **or** $c_p \in \mathcal{D}_p$] {*query failure detector* $\mathcal{D}_p$}

36. **if** [received $(k, c_p, r_p, \mathit{estV}_{c_p}, \mathit{estP}_{c_p})$ from c_p] **then** {p *received* $(\mathit{estV}_{c_p}, \mathit{estP}_{c_p})$ *from* c_p}
37. $\mathit{estV}_p \leftarrow \mathit{estV}_{c_p}$
38. $\mathit{estP}_p \leftarrow \mathit{estP}_{c_p}$
39. $ts_p \leftarrow r_p$
40. send (k, p, r_p, ack) to c_p
41. **else** {p *suspects that* c_p *crashed*}
42. send $(k, p, r_p, nack)$ to c_p
43. **end if**

44. **Phase 4:** {*the current coordinator waits for replies from a majority of processes.*}
{*If those replies indicate that a majority of processes adopted its estimate,*}
{*the coordinator R-broadcasts a decide message*}
45. **if** $p = c_p$ **then**
46. **wait until** [for $\left\lceil \frac{(n+1)}{2} \right\rceil$ processes q : received $(\underline{k}, q, \underline{r_p}, \underline{ack})$ **or** $(\underline{k}, q, \underline{r_p}, \underline{nack})$]
47. **if** [for $\left\lceil \frac{(n+1)}{2} \right\rceil$ processes q : received (k, q, r_p, ack)] **then**
48. *R-broadcast* $(k, p, r_p, \mathit{estV}_p, \mathit{estP}_p, decide)$
49. **end if**
50. **end if**
51. **end while**
52. **end** *LazyConsensus*

53. **when** *R-deliver* $(\underline{k}, q, r_q, \mathit{estV}_q, \mathit{estP}_q, \underline{decide})$ {*if* p *R-delivers a decide msg,* p *decides accordingly*}
54. **if** $state_p = undecided$ **then**
55. $decide(k, \mathit{estV}_q)$
56. $\mathbf{pv}^{k+1} \leftarrow \mathit{estP}_q$ {*updates the permutation vector for the next execution*}
57. $state_p \leftarrow decided$
58. **end if**
59. **end when**

6. SELECTED SCENARIOS FOR SEMI-PASSIVE REPLICATION

Algorithm 2 may seem complex, but most of the complexity is due to the explicit handling of failures and suspicions. So, in order to show that the complexity of the algorithm does not make it inefficient, we illustrate typical executions of the semi-passive replication algorithm based on Lazy Consensus using $\Diamond\mathcal{S}$.

We first present the semi-passive replication in a good run (no failure, no suspicion), as this is the most common case. We then show the execution of the algorithm in the face of a single process crash. Other cases can easily be inferred from these two simple scenarios.

6.1. Semi-Passive Replication in Good Runs

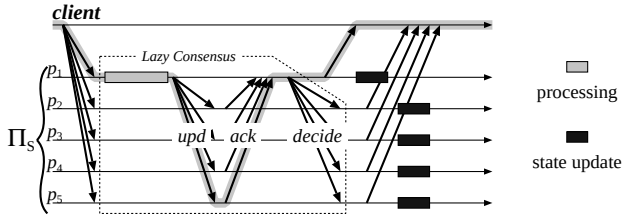

FIG. 7. Semi-passive replication (good run). The critical path request-response is highlighted in gray. The execution of the Lazy Consensus is also depicted in Fig. 5.

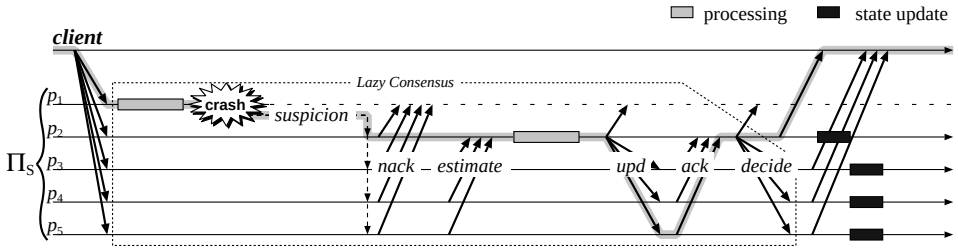

FIG. 8. Semi-passive replication with one failure (worst case). The critical path request-response is highlighted in gray. The execution of the Lazy Consensus in the case of one crash is also depicted in Fig. 6.

We call "good run" a run in which no server process crashes and no failure suspicion is generated. Let Figure 7 represent the execution of Lazy Consensus number k. The server process p_1 is the initial coordinator for consensus k and also the primary. After receiving the request from the client, the primary p_1 handles the request. Once the processing is done, p_1 has the initial value for consensus k. According to the Lazy consensus protocol, p_1 multicasts the update message upd to the backups, and waits for ack messages. Once ack messages have been received (actually from a majority), process p_1 can decide on upd, and multicast the $decide$ message to the backups. As soon as the $decide$ message is received, the servers update their state, and send the reply to the client.

It is noteworthy that the state updates do not appear on the critical path of the client's request (highlighted in gray on the figure).

6.2. Semi-Passive Replication in the Case of One Crash

Figure 8 illustrates the worst case latency for the client in the case of one crash, without incorrect failure suspicions. The worst case scenario happens when the primary p_1 (i.e., the initial coordinator of the Lazy Consensus algorithm) crashes immediately after processing the client request, but before being able to send the update message upd to the backups (compare with Fig. 7). In this case, the communication pattern is different from usual algorithms for passive replication in asynchronous systems, as there is here no membership change.

In more detail, the execution of the Lazy Consensus algorithm runs as follows. If the primary p_1 crashes, then the backups eventually suspect p_1, send a negative acknowledgment message $nack$ to p_1 (the message is needed by the consensus algorithm), and start a new round. The server process p_2 becomes the coordinator for the new round, i.e., becomes the new primary, and waits for $estimate$ messages from a majority of servers: these messages might contain an initial value for the consensus, in which case p_2 does not need to process the client request again. In our worst case scenario, the initial primary p_1 has crashed before being able to multicast the update value upd. So none of the $estimate$ messages

received by p_2 contain an initial value. In order to obtain one, the new primary p_2 processes the request received from the client (Fig. 8), and from that point on, the scenario is similar to the "good run" case of the previous section (compare with Fig. 7).

7. CONCLUSION

Semi-passive replication is a replication technique that does not rely on a group membership for the selection of the primary. While retaining the essential characteristics of passive replication (i.e., non-deterministic processing and parsimonious use of processing resources), semi-passive replication can be solved in an asynchronous system using a $\Diamond\mathcal{S}$ failure detector. This is a significant strength over almost all current systems that implement replication techniques with parsimonious processing. Indeed, in those systems, the replication algorithm requires to force the crash of excluded processes in order to make progress, and thus combines the selection of the primary with the composition of the group.

A second contribution of this paper, Lazy Consensus, is an extension of the Consensus problem to allow the lazy evaluation of process propositions. This means that processes compute their initial value in a "least effort" way, captured with a Laziness property. We have discussed these issues in details in the paper, and presented an algorithm to solve Lazy Consensus. The algorithm was adapted from the Chandra-Toueg consensus algorithm using $\Diamond\mathcal{S}$ [8], and relies on the same assumptions. Even though we have not discussed this issue, other Consensus algorithms could also easily be adapted to solve Lazy Consensus (e.g., [22, 30, 35, 36]).

The semi-passive replication algorithm proposed in this paper is based on solving the problem of Lazy Consensus. The semi-passive replication algorithm however only relies on the conventional properties of Consensus for ensuring the consistency of the replicas. The Laziness property of Lazy Consensus is however the key to the restrained use of resources in semi-passive replication. Depending directly on the quality of failure detectors, the laziness (and hence the parsimony of semi-passive replication) is related to the amount of synchrony exhibited by the system. In particular, in a synchronous system, semi-passive replication ensures that a client request is processed by only one correct replica. Conversely, in the worst case, a single request is never processed by more than about half of the replicas. This behavior is desirable as it naturally allows for a graceful degradation of the replicated service.

We mentioned that semi-passive replication does not require a group membership service, and explained why this is an advantage. This may however give the wrong impression that semi-passive replication is incompatible with a group membership service, or that we believe that such a service is not useful. This is of course not the case, but we regard semi-passive replication as being a lower-level protocol than group membership. Decoupling the replication protocol from housekeeping issues (e.g., releasing resources held by a crashed process, adding or removing processes dynamically) is more elegant and has several advantages in terms of performance, as discussed in [9, 11].

Finally, from the standpoint of clients, our semi-passive replication algorithm is protocol-compatible with active replication. In particular, clients need no specific knowledge about the server replicas, beyond what is necessary to address them as a group. This, combined with the fact that both replication techniques can be implemented based on consensus, makes it much easier for both techniques to coexist. For instance, the use of semi-passive replication in a CORBA Object Group Service made it possible to chose the replication type (active or semi-passive) as a strictly server-side issue and on a per request basis [16].

APPENDIX: PROOFS OF CORRECTNESS

A.1. CORRECTNESS PROOF OF THE SEMI-PASSIVE REPLICATION ALGORITHM

We prove that our algorithm for semi-passive replication (Algorithm 1, page 11) satisfies the properties of the Generic Replication Problem given in Section 3.1. The proof assumes that (1) procedure *LazyConsensus* solves the Lazy Consensus problem according to the specification given in Section 3.2 (ignoring the laziness property[12]), and (2) at least one replica is correct. Solving Lazy Consensus is discussed in Section 5. In fact, Lazy Consensus solves Consensus, which is enough to prove the correctness of the algorithm as a Generic Replication algorithm.

LEMMA 1.1 (Termination). *If a correct client $c \in \Pi_C$ sends a request, it eventually receives a reply.*

Proof. The proof is by contradiction. Let req_c be a request sent by a correct client c that never receives a reply. As c is correct, all correct replicas in Π_S eventually receive req_c at line 10, and insert req_c into their receive queue $recvQ_s$ at line 11. By the assumption that c never gets a reply, no correct replica decides at line 14 on $(req_c, -,)$: if one correct replica would decide, then by the Agreement and Termination property of Lazy Consensus, all correct replicas would decide on $(req_c, -, -)$. As we assume that there is at least one correct replica then, by the property of the reliable channels, and because c is correct, c would eventually receive a reply. Consequently, req_c is never in $hand$ of any replica, and thus no replica s removes req_c from $recvQ_s$ (Hypothesis A).

Let t_0 be the earliest time such that the request req_c has been received by every replica that has not crashed. Let $beforeReqC_s$ denote the prefix of sequence $recvQ_s$ that consists of all requests in $recvQ_s$ that have been received before req_c. After t_0, no new request can be inserted in $recvQ_s$ before req_c, and hence none of the sequences $beforeReqC$ can grow.

Let l be the total number of requests that appear before req_c in the $recvQ$ of any replica:

$$l = \sum_{s \in \Pi_S} \begin{cases} 0 & \text{if } s \text{ has crashed} \\ \#beforeReqC_s & \text{otherwise} \end{cases}$$

After time t_0, the value of l cannot increase since all new request can only be inserted *after* req_c. Besides, after every decision of the Lazy Consensus at line 19, at least one replica s' removes the request $req_{h_{s'}}$ at the head of $recvQ_{s'}$ (l.7, l.22). The request $req_{h_{s'}}$ is necessarily before req_c in $recvQ_{s'}$, and hence belongs to $beforeReqC_{s'}$. As a result, every decision of the Lazy Consensus leads to decreasing the value of l by at least 1.

Since req_c is never removed from $recvQ_s$ (by Hyp. 1.1.1), Task 2 is always enabled ($\#recvQ_s \geq 1$). So, because of the Termination property of Lazy Consensus, the value of l decreases and eventually reaches 0 (this is easily proved by induction on l).

Let t_1 be the earliest time at which there is no request before req_c in the receive queue $recvQ$ of any replica ($l = 0$). This means that, at time t_1, req_c is at the head of the receive queue of all running replicas, and the next execution of Lazy Consensus can only

[12] See Sect. 4.3, p. 11.

decide on request req_c (l.7). Therefore, every correct replica s eventually removes req_c from $recvQ_s$, a contradiction with Hypothesis A. ■

LEMMA 1.2. *For any request req, every replica executes $update(req)$ at most once.*

Proof. Whenever a replica executes $update(req)$ (line 21), it has decided on $(req, -, -)$ at line 15, and inserts req into the set of handled requests $hand$ (line 18). By the Agreement property of Lazy Consensus, every replica that decides at line 15 decides also on $(req, -, -)$ and inserts also req into $hand$ at line 18. As a result, no replica can select req again at line 7, and $(req, -, -)$ cannot be the decision of any subsequent Lazy Consensus. ■

LEMMA 1.3 (Total order). *For any two requests req and req', if some replica executes $update(req')$ after $update(req)$, then a replica executes $update(req')$ only after it has executed $update(req)$.*

Proof. Let req and req' be two requests, and let p be some replica that executes $update(req')$ after it executes $update(req)$. Since p has executed $update(req)$, it has decided $(req, upd_{req}, -)$ at line 19. Let k_1 be the value of variable k when p decides $(req, upd_{req}, -)$. Similarly, p has executed $update(req')$. Let k_2 be the value of variable k when p decides $(req', upd_{req'}, -)$. Because p executes $update(req)$ before it executes $update(req')$, it decides $(req, upd_{req}, -)$ before it decides $(req', upd_{req'}, -)$. Therefore, $k_1 < k_2$.

Let q be any replica that executes $update(req')$. To prove the lemma, we show that q executes $update(req)$ before it executes $update(req')$.

Since q executes $update(req')$, it also decides $(req', upd_{req'}, -)$. Let k'_2 be the value of variable k when it does so. By Lemma 1.2 (at most once) there is only one possible value k'_2. By the Agreement property of Lazy consensus and the fact that p decides $update(req')$ for $k = k_2$, it follows that $k_2 = k'_2$.

If q has decided on the instance k_2 of Lazy consensus, it must have also decided something for $k = k_1$ because $k_1 < k_2$. Again, by the Agreement property of Lazy consensus and the fact that p has decided $(req, upd_{req}, -)$ when $k = k_1$, q has decided $(req, upd_{req}, -)$ when $k = k_1$. By the algorithm, a process executes the $update$ event corresponding to a decision before it starts the next instance of the Lazy consensus. So, because $k_1 < k_2$, process q executes $update(req)$ before it executes $update(req')$. ■

LEMMA 1.4. *If a replica executes $update(req)$, then $send(req)$ was previously executed by a client.*

Proof. If a replica p executes $update(req)$, then some replica q has selected and processed the request req at line 7 and line 8 respectively. It follows that req was previously received by q, as req belongs to the sequence $recvQ_s$. Therefore, req was sent by some client. ■

LEMMA 1.5 (Update integrity). *For any request req, every replica executes $update(req)$ at most once, and only if $send(req)$ was previously executed by a client.*

Proof. The result follows directly from Lemma 1.4 and Lemma 1.2. ■

LEMMA 1.6 (Response integrity). *For any event $receive(resp_{req})$ executed by a client, $update(req)$ is executed by some correct replica.*

Proof. If a client receives $resp_{req}$, then $send(resp_{req})$ was previously executed by some replica (line 16). Therefore, this replica has decided $(req, upd_{req}, res_{req})$ at line 15. By

the Termination and Agreement properties of Lazy Consensus, every correct replica also decides $(req, upd_{req}, res_{req})$ at line 15, and executes $update(req)$ at line 17. The lemma follows from the assumption that at least one replica is correct. ■

THEOREM 1.1. *Algorithm 1 solves the generic replication problem (defined in Section 3.1).*

Proof. Follows directly from Lemma 1.3 (total order), Lemma 1.5 (update integrity), Lemma 1.6 (response integrity), and Lemma 1.1 (termination). ■

A.2. CORRECTNESS PROOF OF THE LAZY CONSENSUS ALGORITHM

Here, we prove the correctness of our Lazy Consensus algorithm (Algorithm 2, page 18). The algorithm solves the weak Lazy Consensus problem using the $\Diamond\mathcal{S}$ failure detector, with a majority of correct processes. Lemma 2.2–2.5 are adapted from the proofs of Chandra and Toueg [8] for the Consensus algorithm with $\Diamond\mathcal{S}$. Without loss of generality and unless specified otherwise, all proofs are expressed for some instance k of the Lazy consensus.

LEMMA 2.1. *No correct process remains blocked forever at one of the* wait *statements.*

Proof. There are three *wait* statements to consider in Algorithm 2 (l.22, l.35, l.46). The proof is by contradiction. Let r be the smallest round number in which some correct process blocks forever at one of the *wait* statements.

In Phase 2, we must consider two cases:

1. If r is the first round, then the current coordinator $c = \mathbf{pv}^k[1]$ does not wait in Phase 2 (l.19), hence it does not block in Phase 2.

2. If $r > 1$ then, all correct processes reach the end of Phase 1 of round r, and they all send a message of the type $(k, -, r, estV, -, -)$ to the current coordinator $c = \mathbf{pv}^k[((r-1) \bmod n) + 1]$ (l.15). Since a majority of the processes are correct, at least $\left\lceil \frac{(n+1)}{2} \right\rceil$ such messages are sent to c and c does not block in Phase 2.

For Phase 3, there are also two cases to consider:

1. c eventually receives $\left\lceil \frac{(n+1)}{2} \right\rceil$ message of the type $(k, -, r, estV, -, -)$ in Phase 2.
2. c crashes.

In the first case, every correct process eventually receives $(k, c, r, estV_c, -)$ (l.35). In the second case, since $\mathcal{D}$ satisfies strong completeness, for every correct process p there is a time after which c is permanently suspected by p, that is, $c \in \mathcal{D}_p$. Thus in either case, no correct process blocks at the second *wait* statement (Phase 3, l.35). So every correct process sends a message of the type $(k, -, r, ack)$ or $(k, -, r, nack)$ to c in Phase 3 (resp. l.40, l.42). Since there are at least $\left\lceil \frac{(n+1)}{2} \right\rceil$ correct processes, c cannot block at the *wait* statement of Phase 4 (l.46). This shows that all correct processes complete round r—a contradiction that completes the proof of the lemma. ■

LEMMA 2.2 (Termination). *Every correct process eventually decides some value.*

Proof. There are two possible cases:

1. **Some correct process decides.** If some correct process decides, then it must have R-delivered some message of type $(k, -, -, -, -, decide)$ (l.53)). By the agreement property of Reliable Broadcast, all correct processes eventually R-deliver this message and decide.

2. **No correct process decides.** Since $\mathcal{D}$ satisfies eventual weak accuracy, there is a correct process q and a time t such that no correct process suspects q after time t. Let $t' \geq t$ be a time such that all faulty processes crash. Note that after time t' no process suspects q. From this and Lemma 2.1, because no correct process decides there must be a round r such that: (i) all correct processes reach round r after time t' (when no process suspects q), and (ii) q is the coordinator of round r (i.e., $q = \mathbf{pv}^k[((r-1) \bmod n) + 1]$). Since q is correct, then it eventually sends a message to all processes at the end of Phase 2 (l.32):

- If $r = 1$ (first round), then q does not wait for any message, and sends $(k, q, r, \mathit{estV}_q, -)$ to all processes at the end of in Phase 2.
- For round $r > 1$, then all correct processes send their estimates to q (l.15). In Phase 2, q receives $\left\lceil \frac{(n+1)}{2} \right\rceil$ such estimates, and sends $(k, q, r, \mathit{estV}_q, -)$ to all processes.

In Phase 3, since q is not suspected by any correct process after time t, every correct process waits for q's estimate (l.35), eventually receives it, and replies with an *ack* to q (l.40). Furthermore, no process sends a *nack* to q (that can only happen when a process suspects q). Thus, in Phase 4, q receives $\left\lceil \frac{(n+1)}{2} \right\rceil$ messages of the type $(k, -, r, ack)$ (and no messages of the type $(k, -, r, nack)$), and q R-broadcasts $(k, q, r, \mathit{estV}_q, -, decide)$ (l.48). By the validity and agreement properties of Reliable Broadcast, eventually all correct processes R-deliver q's message (l.53) and *decide* (l.55)—a contradiction.

So, by Case 2 at least one correct process decides, and by Case 1 all correct processes eventually decide. ■

LEMMA 2.3 (Uniform integrity). *Every process decides at most once.*

Proof. Follows directly from Algorithm 2, where no process decides more than once. ■

LEMMA 2.4 (Uniform agreement). *No two processes decide differently.*

Proof. If no process ever decides, the lemma is trivially true. If any process decides, it must have previously R-delivered a message of the type $(k, -, -, -, -, decide)$ (l.53). By the uniform integrity property of Reliable Broadcast and the algorithm, a coordinator previously R-broadcast this message. This coordinator must have received $\left\lceil \frac{(n+1)}{2} \right\rceil$ messages of the type $(k, -, -, ack)$ in Phase 4 (l.46). Let r be the smallest round number in which $\left\lceil \frac{(n+1)}{2} \right\rceil$ messages of the type $(k, -, r, ack)$ are sent to a coordinator in Phase 3 (l.40). Let c denote the coordinator of round r, that is, $c = \mathbf{pv}^k[((r-1) \bmod n) + 1]$. Let estV_c denote c's estimate at the end of Phase 2 of round r. We claim that for all rounds $r' \geq r$, if a coordinator c' sends $\mathit{estV}_{c'}$ in Phase 2 of round r' (l.32), then $\mathit{estV}_{c'} = \mathit{estV}_c$.

The proof is by induction on the round number. The claim trivially holds for $r' = r$. Now assume that the claim holds for all $r', r \leq r' < x$. Let c_x be the coordinator of round x, that is, $c_x = \mathbf{pv}^k[((x-1) \bmod n) + 1]$. We will show that the claim holds for $r' = x$, that is, if c_x sends estV_{c_x} in Phase 2 of round x (l.32), then $\mathit{estV}_{c_x} = \mathit{estV}_c$.

From Algorithm 2 it is clear that if c_x sends estV_{c_x} in Phase 2 of round x (l.32) then it must have received estimates from at least $\left\lceil \frac{(n+1)}{2} \right\rceil$ processes (l.22).[13] Thus, there is some process p such that (1) p sent a (k, p, r, ack) message to c in Phase 3 of round r (l.40), and (2) the message $(k, p, x, \mathit{estV}_p, -, ts_p)$ is in $msgs_{c_x}[x]$ in Phase 2 of round x (l.23). Since p

[13] Note that $r < x$ hence round x is not the first round.

sent (k, p, r, ack) to c in Phase 3 of round r (l.40), $ts_p = r$ at the end of Phase 3 of round r (l.39). Since ts_p is nondecreasing, $ts_p \geq r$ in Phase 1 of round x. Thus, in Phase 2 of round x, the message $(k, p, x, estV_p, -, ts_p)$ is in $msgs_{c_x}[x]$ with $ts_p \geq r$. It is easy to see that there is no message $(k, q, x, estV_q, -, ts_q)$ in $msgs_{c_x}[x]$ for which $ts_q \geq x$. Let t be the largest ts_q such that message $(k, q, x, estV_q, -, ts_q)$ in $msgs_{c_x}[x]$. Thus, $r \leq t < x$.

In Phase 2 of round x, c_x executes $estV_{c_x} \leftarrow estV_q$ where $(k, q, x, estV_q, -, t)$ is in $msgs_{c_x}[x]$ (l.28). From Algorithm 2, it is clear that q adopted $estV_q$ as its estimate in Phase 3 of round t (l.37). Thus, the coordinator of round t sent $estV_q$ to q in Phase 2 of round t (l.32). Since $r \leq t < x$, by the induction hypothesis, $estV_q = estV_c$. Thus, c_x sets $estV_{c_x} \leftarrow estV_c$ in Phase 2 of round x (l.28). This concludes the proof of the claim.

We now show that, if a process decides a value, then it decides $estV_c$. Suppose that some process p R-delivers $(k, q, r_q, estV_q, -, decide)$, and thus decides $estV_q$. By the uniform integrity property of Reliable Broadcast and the algorithm, process q must have R-broadcast $(k, q, r_q, estV_q, -, decide)$ in Phase 4 of round r_q (l.48). From Algorithm 2, some process q must have received $\left\lceil \frac{(n+1)}{2} \right\rceil$ messages of the type $(k, -, r_q, ack)$ in Phase 4 of round r_q (l.47). By the definition of r, $r \leq r_q$. From the above claim, $estV_q = estV_c$. ■

LEMMA 2.5 (Uniform validity). *If a process decides v, then v was proposed by some process.*

Proof. From Algorithm 2, it is clear that all *estimates* that a coordinator receives in Phase 2 are proposed values. Therefore, the decision value that a coordinator selects from these *estimates* must be the value proposed by some process. Thus, uniform validity of Lazy Consensus is also satisfied. ■

The two properties *proposition integrity* and *weak laziness* are specific to the Lazy Consensus problem. In order to prove them, we first prove some lemmas.

LEMMA 2.6. *Every process that terminates the algorithm considers the same value for the next permutation vector $\mathbf{pv}^{k+1}$ after termination of consensus k.*

Proof. The proof is a trivial adaptation of Lemma 2.4 (uniform agreement) to $estP$ and the fact that $\mathbf{pv}^{k+1}$ is set at line 56. ■

LEMMA 2.7. *Given a sequence of Lazy Consensus problems, processes begin every instance k of the problem with the same permutation vector $\mathbf{pv}^k$.*

Proof. The proof is by induction on the instance number k. Initially, all processes begin the first instance with $k = 1$ and the same permutation vector $\mathbf{pv}^1 = [1, 2, \ldots, n]$, defined to be the identity.

The induction step requires to show that, if all processes begin instance k–1 with the same $\mathbf{pv}^{k-1}$, then they also begin instance k with the same $\mathbf{pv}^k$. This comes as a consequence of Lemma 2.6 and the fact that no process starts an instance before it has completed the previous one. This completes the proof. ■

LEMMA 2.8. *For each process p in Π_S, after p changes its estimate $estV_p$ to a value different from $\perp$, then $estV_p \neq \perp$ is always true.*

Proof. A process p changes the value of its estimate $estV_p$ only at lines 20, 26, 28, and 37. Assuming that $estV_p$ is different from $\perp$, we have to prove that a process p does not set $estV_p$ to $\perp$ if it reaches one of the aforementioned lines.

The result is trivial for lines 20, 26 (by hypothesis the function giv never returns $\perp$) and line 28 (the process selects a value explicitly different from $\perp$).

At line 37, a process sets its estimate to a value received from the coordinator. This value is sent by the coordinator c_p at line 32. Before reaching this line, c_p changed its own estimate $estV_{c_p}$ at one of the following lines: 20, 26, or 28. As shown above, $estV_{c_p}$ is never set to $\perp$ at these lines. ■

LEMMA 2.9. *During a round r, a process p proposes a value only if p is coordinator of round r and estV$_p = \perp$.*

Proof. We say that a process proposes a value when it executes *estV*$_p \leftarrow \mathbf{eval}\ giv$ (line 20 or 26). By line 18, p proposes a value only if p is the coordinator of the round (i.e., $p = c_p$). Let us consider line 20 and line 26 separately.

Line 20: The test at line 19 ensures that line 20 is executed only during the first round. Before executing line 20, $estV_p$ of the coordinator p is trivially equal to $\perp$ (initial value).

Line 26: The result directly follows from the test at line 25. ■

LEMMA 2.10 (Proposition integrity). *Every process proposes a value at most once.*

Proof. We say that a process propose a value when it executes *estV*$_p \leftarrow \mathbf{eval}\ giv$ (lines 20 and 26). We prove the result by contradiction. Assume that some process p proposes a value twice. By definition giv returns a value different from $\perp$. By Lemma 2.8, once $estV_p \neq \perp$, it remains different from $\perp$ forever. By Lemma 2.9, p proposes a value only if $estV_p = \perp$. A contradiction with the fact that p proposes a value twice. ■

LEMMA 2.11. *If two processes p and q propose a value, then at least one of p and q is suspected by a majority of processes in Π_S.*

Proof. We prove this by contradiction. We assume that neither p nor q are suspected by a majority of processes in Π_S. From Lemma 2.9 and the rotating coordinator paradigm (there is only one coordinator in each round), p and q do not propose a value in the same round. Let r_p (resp. r_q) be the round in which p (resp. q) proposes a value. Let us assume, without loss of generality, that p proposes before q ($r_p < r_q$).

During round r_p, any process in Π_S either suspects p or adopts p's estimate (lines 35, 36, 37). Since p is not suspected by a majority of processes in Π_S (assumption), a majority of processes adopt p's estimate. By Lemma 2.8, it follows that (1) a majority of the processes have an estimate different from $\perp$ for any round $r > r_p$.

Consider now round r_q with coordinator q. At line 22, q waits for a majority of estimate messages. From (1), at least one of the estimate messages contains an estimate $estV \neq \perp$. So the test at line 25 returns false, and q does not call giv at line 26. A contradiction with the fact that q proposes a value in round r_q. ■

COROLLARY 2.1 (Weak laziness). *If two processes p and q propose a value, then at least one of p and q is suspected by some processes in Π_S.*

Proof. Follows directly from Lemma 2.11. ■

Lemma 2.11 is obviously not necessary to prove the weak laziness property defined in Section 3.2. However, as stated in Footnote 9 on page 9, it is interesting to show that our algorithm ensures a property stronger than weak laziness. The property is established by Lemma 2.11.

THEOREM 2.1. *Algorithm 2 solves the weak Lazy Consensus problem using $\Diamond\mathcal{S}$ in asynchronous systems with $f = \lfloor \frac{n-1}{2} \rfloor$.*

Proof. Follows directly from Lemma 2.2 (termination), Lemma 2.3 (uniform integrity), Lemma 2.4 (agreement), Lemma 2.5 (validity), Lemma 2.10 (proposition integrity), and Lemma 2.1 (weak laziness). ■

ACKNOWLEDGMENTS

We would like to thank Fernando Pedone for his comments on the specification of replication techniques, and Péter Urbán for his important suggestions that greatly simplified some of the proofs. We are also grateful to the anonymous reviewers for their insightful comments that helped us improve the content of this paper significantly.

This research was in part conducted for the program "Fostering Talent in Emergent Research Fields" in Special Coordination Funds for Promoting Science and Technology by the Japan Ministry of Education, Culture, Sports, Science and Technology.

REFERENCES

1. M. K. Aguilera, W. Chen, and S. Toueg. Using the heartbeat failure detector for quiescent reliable communication and consensus in partitionable networks. *Theor. Comput. Science*, 220(1):3–30, June 1999.
2. M. K. Aguilera, W. Chen, and S. Toueg. Failure detection and consensus in the crash-recovery model. *Distributed Computing*, 13(2):99–125, 2000.
3. M. K. Aguilera, W. Chen, and S. Toueg. On quiescent reliable communication. *SIAM J. Comput.*, 29(6):2040–2073, 2000.
4. M. K. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg. Thrifty generic broadcast. In M. Herlihy, editor, *Proc. 14th Intl. Symp. on Distributed Computing (DISC'00)*, volume 1914 of *LNCS*, pages 268–282, Toledo, Spain, October 2000.
5. P. A. Alsberg and J. D. Day. A principle for resilient sharing of distributed resources. In *Proceedings of the 2nd International Conference on Software Engineering*, pages 562–570, San Francisco, CA, USA, 1976.
6. K. Birman and R. van Renesse, editors. *Reliable Distributed Computing with the Isis Toolkit*. IEEE Computer Society Press, 1994.
7. N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg. The primary-backup approach. In S. Mullender, editor, *Distributed Systems*, ACM Press Books, chapter 8, pages 199–216. Addison-Wesley, second edition, 1993.
8. T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996.
9. B. Charron-Bost, X. Défago, and A. Schiper. Time vs. space in fault-tolerant distributed systems. In *Proc. 6th IEEE Intl. Workshop on Object-oriented Real-time Dependable Systems (WORDS'01)*, Rome, Italy, January 2001.
10. G. Chockler, I. Keidar, and R. Vitenberg. Group communication specifications: A comprehensive study. *ACM Comput. Surv.*, 33(4):427–469, December 2001.
11. X. Défago, P. Felber, and A. Schiper. Optimization techniques for replicating CORBA objects. In *Proc. 4th IEEE Intl. Workshop on Object-oriented Real-time Dependable Systems (WORDS'99)*, pages 2–8, Santa Barbara, CA, USA, January 1999.
12. X. Défago, A. Schiper, and N. Sergent. Semi-passive replication. In *Proc. 17th IEEE Intl. Symp. Reliable Distributed Systems (SRDS'98)*, pages 43–50, West Lafayette, IN, USA, October 1998.
13. X. Défago, A. Schiper, and P. Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. RR IS-RR-2003-009, Japan Advanced Institute of Science and Technology, Ishikawa, Japan, September 2003.
14. Xavier Défago. *Agreement-Related Problems: From Semi-Passive Replication to Totally Ordered Broadcast*. PhD thesis, École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland, August 2000. Number 2229.
15. D. Essamé, J. Arlat, and D. Powell. PADRE: a protocol for asymmetric duplex redundancy. In *IFIP 7th Working Conference on Dependable Computing in Critical Applications (DCCA-7)*, pages 213–232, San Jose, CA, USA, January 1999.
16. P. Felber, X. Défago, P. Eugster, and A. Schiper. Replicating CORBA objects: a marriage between active and passive replication. In *Proc. 2nd IFIP Intl. Working Conf. on Distributed Applications and Interoperable Systems (DAIS'99)*, pages 375–387, Helsinki, Finland, June 1999.

17. C. Fetzer. Perfect failure detection in timed asynchronous systems. *IEEE Trans. on Computers*, 52(2):99–112, February 2003.

18. M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, April 1985.

19. U. Fritzke, P. Ingels, A. Mostéfaoui, and M. Raynal. Consensus-based fault-tolerant total order multicast. 12(2):147–156, February 2001.

20. D. Gries and F. B. Schneider. *A Logical Approach to Discrete Math*. Texts and monographs in computer science. Springer-Verlag, 1993.

21. R. Guerraoui and A. Schiper. Software-based replication for fault tolerance. *IEEE Computer*, 30(4):68–74, April 1997.

22. M. Hurfin, R. Macêdo, M. Raynal, and F. Tronel. A general framework to solve agreement problems. In *Proc. 18th IEEE Intl. Symp. on Reliable Distributed Systems (SRDS'99)*, pages 56–67, Lausanne, Switzerland, October 1999.

23. M. Hurfin, A. Mostéfaoui, and M. Raynal. Consensus in asynchronous systems where processes can crash and recover. In *Proc. 17th IEEE Intl. Symp. Reliable Distributed Systems (SRDS'98)*, pages 280–286, West Lafayette, IN, USA, October 1998.

24. L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, R. K. Budhia, C. A. Lingley-Papadopoulis, and T. P. Archambault. The Totem system. In *Proc. 25rd Intl. Symp. on Fault-Tolerant Computing (FTCS-25)*, pages 61–66, Pasadena, CA, USA, 1995.

25. R. Oliveira, R. Guerraoui, and A. Schiper. Consensus in the crash-recover model. TR 97/239, École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland, August 1997.

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

27. S. Pleisch and A. Schiper. Modeling fault-tolerant mobile agent execution as a sequence of agreement problems. In *Proc. 19th IEEE Intl. Symp. on Reliable Distributed Systems (SRDS'00)*, pages 11–20, Nürnberg, Germany, October 2000.

28. S. Poledna. Replica determinism in distributed real-time systems: A brief survey. *Real-Time Systems*, 6(3):289–316, May 1994.

29. D. Powell. *Delta4: A Generic Architecture for Dependable Distributed Computing*, volume 1 of *ESPRIT Research Reports*. Springer-Verlag, 1991.

30. A. Schiper. Early consensus in an asynchronous system with a weak failure detector. *Distributed Computing*, 10(3):149–157, 1997.

31. F. B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv.*, 22(4):299–319, December 1990.

32. J. B. Sussman and K. Marzullo. Comparing primary-backup and state machines for crash failures. In *Proc. 15th ACM Symp. on Principles of Distributed Computing (PODC-15)*, page 90, Philadelphia, PA, USA, May 1996. Brief announcement.

33. P. Urbán, I. Shnayderman, and A. Schiper. Comparison of failure detectors and group membership: Performance study of two atomic broadcast algorithms. In *Proc. IEEE Intl. Conf. on Dependable Systems and Networks (DSN'03)*, pages 645–654, San Francisco, CA, USA, June 2003.

34. R. van Renesse, K. P. Birman, and R. Cooper. The HORUS system. TR, Dept. of Computer Science, Cornell University, Ithaca, NY, USA, 1993.

35. C. Yahata, J. Sakai, and M. Takizawa. Generalization of consensus protocols. In *Proc. 9th Intl. Conf. on Information Networking (ICOIN-9)*, pages 419–424, Osaka, Japan, 1994.

36. C. Yahata and M. Takizawa. General protocols for consensus in distributed systems. In *Proc. 6th Intl. Conf. on Database and Expert Systems Applications (DEXA'95)*, number 978 in LNCS, pages 227–236, London, UK, September 1995. Springer-Verlag.

37. H. Zou and F. Jahanian. Real-time primary-backup replications with temporal consistency. In *Proc. 18th IEEE Intl. Conf. on Distributed Computing Systems (ICDCS-18)*, pages 48–56, Amsterdam, The Netherlands, May 1998.