Specification of Replication Techniques, Semi-Passive Replication, and Lazy Consensus[*]

Xavier Défago

Graduate School of Knowledge Science,
Japan Advanced Institute of Science and Technology,
1-1 Asahidai, Tatsunokuchi, Ishikawa 923-1292,
Japan

André Schiper

School of Computer and Communication Sciences,
Swiss Federal Institute of Technology in Lausanne,
CH-1015 Lausanne EPFL,
Switzerland

Abstract

This paper brings the following three main contributions: a hierarchy of specifications for replication techniques, semi-passive replication, and Lazy Consensus.

Based on the definition of the *Generic Replication problem*, we define two families of replication techniques: replication with parsimonious processing (e.g., passive replication), and replication with redundant processing (e.g., active replication). This helps relate replication techniques to each other.

We define a novel replication technique with parsimonious processing, called semi-passive replication, for which we also give an algorithm. The most significant aspect of semi-passive replication is that it requires a weaker system model than existing techniques of the same family.

We define a variant of the Consensus problem, called Lazy Consensus, upon which our semi-passive replication algorithm is based. The main difference between Consensus and Lazy Consensus is a property of laziness which requires that initial values are computed only when they are actually needed.

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

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. However, the distributed nature of those systems provides the mean to *increase* their reliability. The 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 [18]. 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 [22, 30], each request is processed by all replicas. 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 of non-determinism [27]. For instance, multi-threading typically introduces non-determinism.

With passive replication (also called *primary-backup*) [9, 18], only one replica (primary) processes the request, and sends update messages to the other replicas (backups). This technique 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

[0] A preliminary abstract of this paper appeared in *Proceedings of the 17th IEEE International Symposium on Reliable Distributed Systems* (IEEE CS Press, pp. 43–50). It is now largely superseded by this version.

[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.

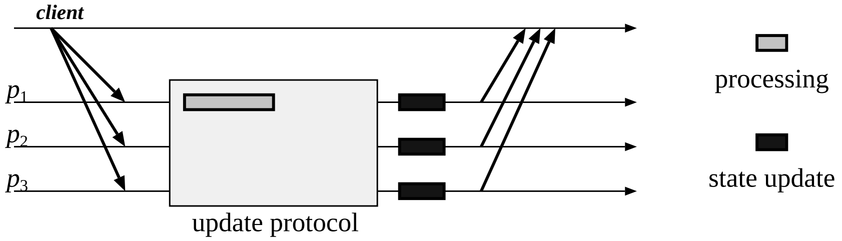

Figure 1: Semi-passive replication (no crash).
(conceptual representation: the *update protocol* actually hides several messages)

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 [4, 9, 15, 28, 36]. 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 suspected to have crashed (e.g., [6, 24, 33]). In practice, this is a strong limitation of passive replication since this means that a mere suspicion will 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 [10]).

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.

The main contribution of this paper is to specify the problem of semi-passive replication, and to propose an algorithm to solve it. We also give a general definition for replication techniques, and prove the correctness of our semi-passive replication algorithm with respect to this definition. Our semi-passive algorithm is based on a variant of the Consensus problem called Lazy Consensus, for which we also give a specification, an algorithm, and proofs of correctness.

The rest of the paper is structured as follows. Section 2 gives a brief informal overview of semi-passive replication. Section 3 presents the system model considered in this paper, and defines the notation used by the algorithms. Section 4 proposes a hierarchy of problems that define replication techniques. In Section 5, we give an algorithm for semi-passive replication, and specify the problem of Lazy Consensus used by this algorithm. In Section 6 we present an algorithm for the Lazy Consensus problem designed for asynchronous systems augmented with a failure detector. Section 7 illustrates the execution of our semi-passive replication algorithm. Section 8 concludes the paper.

2 Overview of Semi-Passive Replication

The passive replication technique is quite useful in practice, since it requires less processing power than active replication and makes no assumption on the determinism of processing a request. Semi-passive replication can be seen as a variant of passive replication, as it retains most of its major characteristics (e.g., allows for non-deterministic processing). However, in semi-passive replication, the selection of the primary is based on the rotating coordinator paradigm [10] and not on a group membership service as usually done in passive replication. The rotating coordinator mechanism is a much simpler mechanism.

Informally, semi-passive replication works the following way. 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

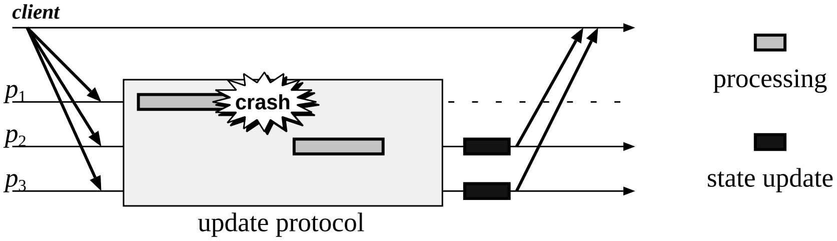

Figure 2: Semi-passive replication (crash of the coordinator).
(conceptual representation: the *update protocol* actually hides several messages)

later in Section 5. 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 5. 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.

3 System Model and Definitions

We define here some notation used in the paper. Replication techniques are usually defined in the client-server interaction model. We describe this model and introduce some related notations.

3.1 Client-Server Model

We consider two types of processes: (1) clients and (2) replicas of a server. 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] We also denote the number of server processes by $n = |\Pi_S|$.

We assume that the system is asynchronous (i.e., there is no bound on communication delays and on the relative speed of processes). Processes fail by crashing, i.e., we do not consider malicious (also known as Byzantine) processes. A correct process is a process that does not crash. Crashes are permanent.[3] Processes communicate through quasi-reliable communication channels, that is, if a correct process p sends a message m to a correct process q, then q eventually receives m.

3.2 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 Gries and Schneider [17].

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 [17]) 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

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

[2] Note that Π_C and Π_S may overlap.

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

□ send(req) ■ receive(resp) ▨ handle(req) ▩ update(req)

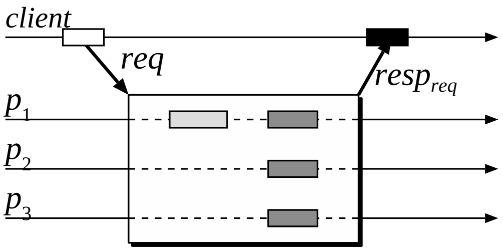

Figure 3: Replication with parsimonious processing

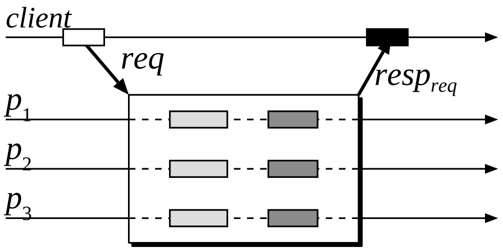

Figure 4: Replication with redundant processing

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.

4 Specification of Replication Techniques

In the context of replication techniques, there exist specifications for passive replication [9, 8] and for active replication [22, 30]. However, these specifications are expressed in such a way that they are not compatible with each other. This is strongly against the intuition that both replication techniques solve a common problem: replication.

In this section, we define this common problem, as the *Generic Replication Problem*. We then define various replication techniques—passive, semi-passive, and active replication—by extending the specification of the Replication problem. This defines a hierarchy of replication problems, and thus brings a new light on the relation between those problems.

4.1 The Generic Replication Problem

Without loss of generality, the Generic Replication Problem is defined in the client-server model. The clients issue requests to a (replicated) server, and receive replies. We consider that a server is replicated when it is composed of more than one process (called replicas hereafter). The server replicas update their respective state according to the requests that they receive from the clients.

We consider a model in which each process is modeled as a state machine. There are two types of state machines: 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. The Generic Replication Problem is defined by the following properties:

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

(TOTAL ORDER) If there is an event $update(req)$ such that a server replica executes $update(req)$ as its i^{th} update event, then all server replicas that execute the i^{th} update event execute $update(req)$ — as their i^{th} update event.

(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.

Replication techniques satisfy these properties. Specific replication techniques can be grouped into two basic families that differ only with the fact that they satisfy additional properties: replication techniques with parsimonious processing, and replication techniques with redundant processing. The replication techniques with *parsimonious* processing (see Fig. 3) are replication techniques that restrict the processing of requests to a single process. This requirement ensures that processing requests consumes as few resources as possible. Conversely, the replication techniques with *redundant* processing (see Fig. 4) require that all processes process each request.

4.2 Replication Techniques with Parsimonious Processing

Replication techniques with parsimonious processing satisfy a property of *parsimony*, which states that requests are normally processed by one replica only. The various replication techniques that belong to this family differ by the exact definition of this property. We define three different parsimony properties, and present then three replication techniques with parsimonious processing: *passive replication*, *coordinator-cohort*, and *semi-passive replication*.

4.2.1 Parsimony Properties

We distinguish between three parsimony properties: *strong parsimony*, *quasi-strong parsimony* and *weak parsimony*.

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

Strong parsimony forbids to ever suspect a process before it crashes. This requires the assumption of either a synchronous system model [19, 23, 5] or an asynchronous system model with a perfect failure detector (known as class $\mathcal{P}$) [10].

(QUASI-STRONG PARSIMONY) If the same request req is processed by two replicas p and q, then p and q are not both correct.

Quasi-strong parsimony is weaker than strong parsimony, because unlike strong parsimony, quasi-strong parsimony is not violated in the following case: two replicas p and q process the same request req, and p crashes after q has processed the request. In practice, quasi-strong parsimony can be ensured in the context of the (primary partition) group membership service [7]. With such a service a process, e.g., p in the example above, learning that it has been incorrectly suspected and excluded from the group, kills himself. [4]

(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.

The definition of weak parsimony links parsimony to the properties of a failure detector (which outputs lists of suspected processes). It is important to understand the fundamental difference between weak parsimony and quasi-strong parsimony. If two correct processes p and q process the same request then quasi-strong parsimony can only be satisfied if either one of the two processes is forced to crash. Conversely, weak parsimony never forces any of the processes to crash.

4.2.2 Passive Replication

Passive replication, also sometimes called primary-backup [9], requires to select one particular replica called the *primary*. Clients issue their requests to the primary. The primary processes every request and sends update messages to the other replicas, called the *backups*. Once the backups have been brought up-to-date, the primary sends a reply to the client. If the primary crashes during the processing of a request, it is the client's responsibility to reissue its request to a new primary.

Passive replication can be characterized as satisfying the *strong parsimony* property defined above: [5] after a replica p (primary) has processed the request req, then the same request can be processed by another replica q (new primary) only after p has crashed.

[4] See [14] for a discussion on process exclusion and process controlled crash.

[5] Linking passive replication to the strong parsimony property is specific to this paper. However, most systems that implement passive replication assume a synchronous system. So, we believe that linking passive replication to the strong parsimony property makes sense and helps clarifying the specificity of the different replication techniques.

4.2.3 Coordinator-Cohort

Coordinator-cohort [7] is a variant of passive replication designed in the context of the Isis group communication system [6]. With this replication scheme, one of the replicas is designated as the *coordinator* (i.e., primary) and the other replicas as the *cohort* (i.e., backups). Unlike passive replication, a client sends its request to the whole group of replicas. Then, both techniques behave similarly: the coordinator processes the request and sends update messages to the cohort. If the coordinator crashes, a new coordinator is selected by the group and takes over the processing of the request.

In addition to a different interaction model, coordinator-cohort bases the selection of the coordinator on a group membership service. As a result, coordinator-cohort guarantees *quasi-strong parsimony* rather than *strong parsimony*.

4.2.4 Semi-Passive Replication

In this paper, we introduce the semi-passive replication technique. This new replication technique bears similarities with passive replication and coordinator-cohort, but differs by several important aspects. The interaction model resembles that of coordinator-cohort. The most important difference is that, contrary to coordinator-cohort, semi-passive replication, which satisfies weak parsimony, never forces any of the processes to crash.

4.3 Replication Techniques with Redundant Processing

Replication techniques with redundant processing require that all correct replicas process each request. This is clearly in contrast with the requirement of parsimonious processing set forth by the replication techniques presented in Section 4.2. The main motivation for redundant processing is to guarantee a constant response time, regardless of the occurrence of failures. This is hence no surprise that this family of replication techniques is usually favored by real-time systems designers [27]. There are two main replication techniques with redundant processing: *active replication*, and *semi-active replication*.

4.3.1 Active Replication

In active replication [22, 30], clients sends their requests to all replicas. All replicas process incoming requests in the same way and in the same order. So, not only the state of the replicas is replicated, but also the processing of the requests. To reflect this, active replication must ensure the following property of *redundancy*:

(REDUNDANCY) For any request req, if a process p executes the event $update(req)$, then p has previously executed the event $handle(req)$.

In practice, the processing of the requests (i.e., $handle(req)$) is interleaved with the application of the state updates (i.e., $update(req)$), as illustrated on Figure 4.

4.3.2 Semi-Active Replication

Semi-active replication [28] is a replication technique with redundant processing that improves on active replication by circumventing the requirement for deterministic processing. Semi-active replication extends active replication with the notion of *leader* and *followers*. While the actual processing of a request is performed by all replicas, it is the responsibility of the leader to perform the non-deterministic parts of the processing and inform the followers.

The semi-active replication technique was implemented in the context of Delta-4 [28]. The project was designed to support the replication of time-critical applications, and assumes a synchronous system model.

As an extension of active replication, semi-active replication must also satisfy a property of redundancy. However, the specification must be extended to define the exact role of leader and followers, and hence explicitly lift the constraint of deterministic processing. Such a definition is however beyond the scope of this paper.

5 Semi-Passive Replication Algorithm

We begin this section by giving a general overview of the semi-passive replication algorithm. First, we introduce the problem of *Lazy Consensus*, on which our semi-passive replication algorithm is based. We then present our

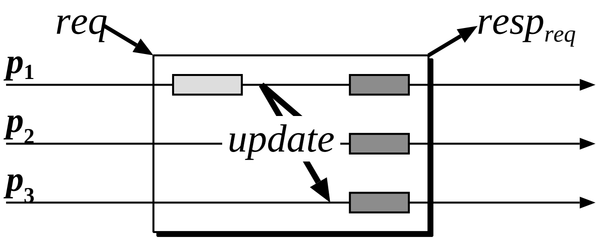

Figure 5: Semi-passive replication: update message sent by the primary.

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 algorithm is proved in the appendix).

5.1 Basic Idea: Consensus on "update" values

As mentioned in Section 2, 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 5.

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 (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 the question of selection a primary inside the consensus algorithm. A process p takes the role of the primary (i.e., handles client requests) exactly when it proposes its initial value for Consensus.

5.2 Lazy Consensus

The Lazy Consensus problem is a generalization of the Consensus [10] problem: it allows processes to delay the computation of their initial value — in the standard Consensus problem each process starts with an initial value; in the Lazy Consensus each process gets its initial value only when necessary.[6]

5.2.1 Specification of Lazy Consensus

We consider the set of processes Π_S. With the standard Consensus [10], processes in Π_S begin by proposing a value. This is in contradiction with the laziness property that we want to enforce. So, 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 $v \neq \bot$[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.

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

[7] giv stands for *get initial value*.

[8] The symbol $\bot$ (bottom) is a common way to denote the absence of value. This is called either *nil* or *null* in most programming languages.

(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. In the sequel, we present an algorithm for semi-passive replication that uses Lazy Consensus. Solving Lazy Consensus is discussed in Section 6.

Remark 1 *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 or q has crashed before p proposes a value.*

Note that the strong laziness property can only be ensured in a system in which failures are always correctly detected (e.g., synchronous system, asynchronous system with a perfect failure detector).

With weak laziness, an incorrect suspicion may lead two correct processes to propose a value. The overhead is related to the price of generating the initial value. In practice, if failure detectors are tuned properly, the probability of incorrectly suspecting a process should remain low.

5.3 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. 5.4, p. 9). However, laziness does not affect the correctness of the algorithm as a *Generic Replication* problem (see Sect. A, p. 18; Remark 2, p. 10)

5.3.1 Notation and System Model

In order to express our semi-passive replication algorithm, we introduce the following notation.

- 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 $resp_{req}$.
- $update : (upd_{req}, state_{s'}) \longmapsto state'_{s'}$: 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 in Section 4, where s' is the server that executes *update*.

We express the semi-passive replication algorithm as tasks that can execute concurrently (Algorithm 1, page 9). A task with a clause **when** is enabled when the condition is true. We assume that (1) any task that is enabled is eventually executed, and (2) there are no multiple concurrent executions of the same task (Task 1 or Task 2 in Algorithm 1). We also make the following assumptions regarding the system (see Sect. 3.1):

- processes fail by crashing only (no Byzantine failures);
- communication channels are quasi-reliable (no message loss if sender and destination are correct);
- crashed processes never recover (see Footnote 3, p. 3).

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 [1, 3], and to handle process recovery [2, 21, 26]. However, this would obscure the key idea of semi-passive replication by introducing unnecessary complexity.

[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 is suspected by a *majority* of processes in Π_S (Lemma 22, p. 22).

5.3.2 Algorithm

We now give the full algorithm (Algorithm 1 below), which solves semi-passive replication as a sequence of Lazy Consensus problems. The algorithm is executed by every server process.

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.

Lines 11 – 15 Whenever s receives a new request (line 10), it is appended to $recvQ_s$ (line 12).

Lines 13 – 20 This is the main part of the algorithm. A new Lazy Consensus is started whenever the preceding one terminates, and the receive queue $recvQ_s$ is not empty (line 13). At the end of the consensus, the request that has been processed (handling and update) is removed from $recvQ_s$ and inserted into $hand_s$ (line 20). At line 15 the Lazy Consensus algorithm is started. Then, the server waits for the decision value $(req, upd_{req}, resp_{req})$ of Consensus: req is the request that has been handled; upd_{req} is the update resulting from handling req; and $resp_{req}$ is the response that should be sent to the client.

At line 17, the response $resp_{req}$ is sent to the client. At line 18, the local state of the server s is updated according to the update message upd_{req}. Finally, the request that has been handled is inserted into the set $hand$ at line 20.

Lines 6 – 10: The function $handleRequest$ The function *handleRequest* returns initial values for each instance of the Lazy consensus problem. The function *handleRequest* (line 6) is called by the Lazy consensus algorithm. In other words, when a process calls *handleRequest*, it acts as a primary in the context of the semi-passive replication technique.

The function *handleRequest* selects a client request that has not been handled yet (line 7), handles the request (line 8), and returns the selected request req, the update message upd_{req} resulting from handling req, as well as the corresponding response message $resp_{req}$ (line 9).

Algorithm 1 Semi-passive replication (code of server s)

```
 1: Initialisation:
 2:    $recvQ_s \leftarrow \epsilon$                      {sequence of received requests, initially empty}
 3:    $hand_s \leftarrow \emptyset$                                          {set of handled requests}
 4:    $state_s \leftarrow state^0$
 5:    $k \leftarrow 0$

 6: function handleRequest()
 7:    $req \leftarrow head.recvQ_s$
 8:    $(upd_{req}, resp_{req}) \leftarrow handle(req, state_s)$
 9:    return $(req, upd_{req}, resp_{req})$

10: when receive($req_c$) from client $c$                                         {Task 1}
11:    if $req_c \notin hand_s \wedge req_c \notin recvQ_s$ then
12:       $recvQ_s \leftarrow recvQ_s \triangleright req_c$

13: when $\#recvQ_s > 0$                                                           {Task 2}
14:    $k \leftarrow k + 1$
15:    LazyConsensus($k$,handleRequest)                           {Solve the $k^{th}$ Lazy consensus}
16:    wait until $(req, upd_{req}, resp_{req}) \leftarrow decided$
17:    send $(resp_{req})$ to $sender(req)$                                {Send response to client}
18:    $state_s \leftarrow update(upd_{req}, state_s)$                              {Update the state}
19:    $recvQ_s \leftarrow recvQ_s - \{req\}$
20:    $hand_s \leftarrow hand_s \cup \{req\}$
```

5.4 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 Section 4.1 (but it might not satisfy the *parsimonious processing* property, Sect. 4.2.1).

THEOREM 11. Algorithm 1 solves the generic replication problem (defined in Section 4.1).

The details of the proof are given in the appendix (pp. 18–19). It is nevertheless important to note that Theorem 11 is proved independently of the laziness property of the Consensus.

LEMMA 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 2. *Algorithm 1 with* weak *Lazy Consensus solves the* semi-passive replication *problem.*

PROOF. Follows directly from Theorem 11 (generic replication) and Lemma 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. 5.2.1).

LEMMA 3. *Algorithm 1 with* strong *Lazy Consensus satisfies* strong *parsimony.*

PROOF. The proof is a trivial adaptation from that of Lemma 3. □

COROLLARY 4. *Algorithm 1 with* strong *Lazy Consensus solves the* passive replication *problem.*

PROOF. Follows directly from Theorem 11 (generic replication) and Lemma 3 (strong parsimony). □

Remark 2 *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 orthogonal.

6 Solving Lazy Consensus

In this section, we give an algorithm that solves the problem of Lazy Consensus defined in Section 5.2.1. [10] The algorithm is based on the asynchronous model augmented with failure detectors [10].

We first give a brief description of the system model for solving Lazy Consensus and we give an algorithm that solves the Lazy Consensus problem using the class $\Diamond\mathcal{S}$ of failure detectors.

6.1 System Model

In order to solve Lazy Consensus among the server processes Π_S, we consider an asynchronous system augmented with failure detectors [10]. Semi-passive replication could easily be expressed in other system models in which Lazy Consensus is solvable. We have chosen the failure detector model for its generality and its conceptual simplicity. We assume here the $\Diamond\mathcal{S}$ failure detector defined on the set of server processes Π_S, which is defined by the following properties [10]:

(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.

[10] An earlier version of this algorithm was called $\mathcal{DIV}$consensus [13]. Note that $\mathcal{DIV}$consensus used to designate an *algorithm*, whereas Lazy Consensus now designates a *problem*.

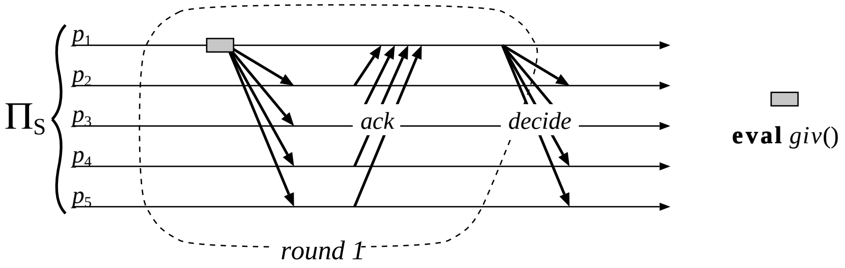

Figure 6: Lazy Consensus with $\Diamond\mathcal{S}$ (no crash, no suspicion)

6.2 Lazy Consensus Algorithm using $\Diamond\mathcal{S}$

The algorithm for Lazy Consensus is adapted from the consensus algorithm using $\Diamond\mathcal{S}$ proposed by Chandra and Toueg [10]. This algorithm assumes that a majority of the processes are correct ($f = \lfloor \frac{n}{2} \rfloor$, where f is the maximum number of processes that may crash). The algorithm proceeds in asynchronous rounds, and is based on the rotating coordinator paradigm: in every round another process is coordinator.

Note that different algorithms for Lazy Consensus using $\Diamond\mathcal{S}$ can be adapted from other consensus algorithms based on the rotating coordinator paradigm (e.g., [25, 29]).

6.2.1 Solving the Lazy Consensus Problem

Algorithm 2 (page 13) solves the Lazy Consensus problem. There are actually only little difference between the Algorithm 2 and the $\Diamond\mathcal{S}$ consensus algorithm of [10]: the lines in which the two algorithms differ are highlighted with an arrow in the margin on Algorithm 2. We do not give a full explanation of the algorithm as the details can be found in [10]. However, we explain the parts on which the algorithms differ. Compared with Chandra and Toueg's algorithm, we have the following three differences.

1. *Laziness (Alg. 2, lines 4,19,24–27).* The most important difference is the modifications necessary to satisfy the laziness property, see Sect. 6.2.2.

2. *Reordering of the process list (Alg. 2, lines 2,5,10,28,37,53).* The ability of the algorithm to reorder the list of processes improves its efficiency in the case of a crash (in the case of a sequence of consensus executions), see Sect. 6.2.3.

3. *Optimization of the first phase (Alg. 2, lines 13,18).* The first phase of the first round of the consensus algorithm is not necessary for the correctness of the algorithm [12, 29]. We have thus optimized it away, see Sect. 6.2.4.

6.2.2 Laziness

In the context of a solution based on the rotating coordinator paradigm, a process p_i proposes a value only when it is coordinator, and only if p_i is not aware of any previously computed initial value. So, in the absence of failures (and suspicions), only one process calls the function giv (see Sect. 5.2.1). Figure 6 illustrates a run of the algorithm in the absence of crash (and incorrect failure suspicions). Only the coordinator of the first round (process p_1) calls the function giv, and the Lazy Consensus problem is solved in one round. If p_1 crashes, two rounds may be needed to solve the problem (see "consensus k" on Fig. 7): process p_1, coordinator of the first round, calls giv and crashes. The surviving processes move to the second round. Process p_2, coordinator of the second round, in turn calls giv.

6.2.3 Reordering of the System List

In the rotating coordinator paradigm, every instance of the Chandra and Toueg's Consensus algorithm invariably starts the first round by selecting the same process, say p_1, as the coordinator. If p_1 crashes, further executions of the consensus algorithm will always require at least two rounds to complete. This extra cost (two rounds instead of one) is avoided by a very simple idea.

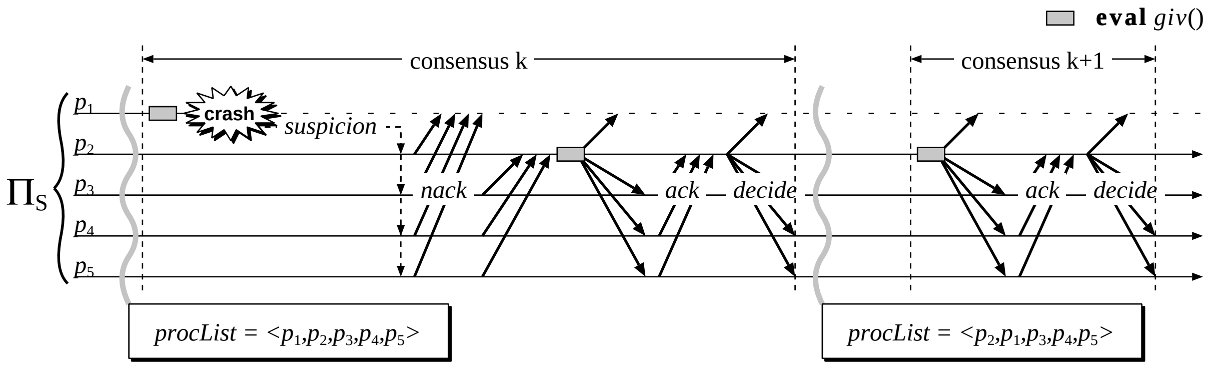

Figure 7: Permutations of Π_S and selection of the coordinator

Assume that, for the consensus number k, the processes of Π_S are ordered as follows: $[p_1, p_2, p_3, p_4, p_5]$, which defines p_1 as the first coordinator (see Fig. 7). If p_1 is suspected (e.g., it has crashed) during consensus k, the processes of Π_S are reordered $[p_2, p_3, p_4, p_5, p_1]$ for the consensus $k+1$, which defines p_2 as the first coordinator. So, despite the crash of p_1 in consensus k, consensus $k+1$ can be solved in one single round.

This reordering, implemented in the context of the Lazy Consensus algorithm, requires no additional message. In the algorithm, processes not only try to reach an agreement on the decision value, but also on the order of the process list. For this purpose, they manage two estimate variables: $estV_p$ for the decision value, and $estL_p$ for the process list. When a coordinator proposes a value, it also proposes a process list in which it is the first coordinator (see Alg. 2, line 5 and 31).

Remark 3 *Note that we present the idea of the dynamic process list using a simple reordering policy. This is enough to illustrate the idea but it is possible, in practice, to reorder the list according to some smarter heuristics. Changing the reordering policy does not compromise the correctness of the algorithms, as long as the list is only reordered at line 5 and 31 in Algorithm 2.*

Remark 4 *Because of its name, the process list managed by our Lazy Consensus algorithm could easily be mistaken for some kind of group membership. There is however a fundamental difference: a group membership defines (and dynamically modifies) a set of processes, whereas the process list employed by our algorithm defines (and dynamically modifies) an order relation on a given (static) set of processes. In other words, the membership never changes in the latter case.*

6.2.4 Optimization of the First Phase

Compared with Chandra and Toueg's algorithm, a further difference consists in the optimization of the first phase of the algorithm [29]. In the Lazy Consensus algorithm, all processes start with $\perp$ 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 in line 19.

7 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 the context of a good run (no failure, no suspicion), as this is the most common case. We then show the execution of the algorithm in the context of one process crash.

7.1 Semi-Passive Replication in Good Runs

We call "good run" a run in which no server process crashes and no failure suspicion is generated. Let Figure 8 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.

Algorithm 2 Lazy Consensus (code of process p)

```
  Initialisation:
→     $procList_p \leftarrow \langle p_1, p_2, \ldots, p_n \rangle$

  **procedure** *LazyConsensus* (**function** $giv : \emptyset \mapsto v$)
→     $estV_p \leftarrow \bot$    $\{$*p's estimate of the decision value*$\}$
→     $estL_p \leftarrow p \lhd (procList_p - \{p\})$    $\{$*p's estimate of the new process list (with p at the head)*$\}$
      $state_p \leftarrow undecided$
      $r_p \leftarrow 0$    $\{r_p$ *is p's current round number*$\}$
      $ts_p \leftarrow 0$    $\{ts_p$ *is the last round in which p updated estV$_p$, initially 0*$\}$

      **while** $state_p = undecided$ **do do**    $\{$*rotate through coordinators until decision reached*$\}$
→         $c_p \leftarrow procList_p\,[(r_p \bmod n) + 1]$    $\{c_p$ *is the current coordinator*$\}$
          $r_p \leftarrow r_p + 1$

          **Phase 1:**    $\{$*all processes p send estV$_p$ to the current coordinator*$\}$
→             **if** $r_p > 1$ **then**
                  send $(p, r_p, estV_p, estL_p, ts_p)$ to $c_p$
→             **end if**

          **Phase 2:**    $\{$*coordinator gathers* $\left\lceil \frac{(n+1)}{2} \right\rceil$ *estimates and proposes new estimate*$\}$
              **if** $p = c_p$ **then**
→                 **if** $r_p = 1$ **then**
→                     $estV_p \leftarrow$ **eval** $giv()$    $\{p$ proposes *a value*$\}$
→                 **else**
                      **wait until** [for $\left\lceil \frac{(n+1)}{2} \right\rceil$ processes $q$: received $(q, r_p, estV_q, estL_q, ts_q)$ from $q$]
                      $msgs_p[r_p] \leftarrow \{(q, r_p, estV_q, estL_q, ts_q) \mid p$ received $(q, r_p, estV_q, estL_q, ts_q)$ from $q\}$
                      $t \leftarrow$ largest $ts_q$ such that $(q, r_p, estV_q, estL_q, ts_q) \in msgs_p[r_p]$
→                     **if** $estV_p = \bot$ **and** $\forall (q, r_p, estV_q, estL_q, ts_q) \in msgs_p[r_p] : estV_q = \bot$ **then**
→                         $estV_p \leftarrow$ **eval** $giv()$    $\{p$ proposes *a value*$\}$
→                     **else**
→                         $estV_p \leftarrow$ select one $estV_q \neq \bot$ s.t. $(q, r_p, estV_q, estL_q, t) \in msgs_p[r_p]$
→                         $estL_p \leftarrow estL_q$
→                     **end if**
→                 **end if**
→                 send $(p, r_p, estV_p, estL_p)$ to all
              **end if**

          **Phase 3:**    $\{$*all processes wait for new estimate proposed by current coordinator*$\}$
              **wait until** [received $(c_p, r_p, estV_{c_p}, estL_{c_p})$ from $c_p$ **or** $c_p \in \mathcal{D}_p$]    $\{$*query failure detector* $\mathcal{D}_p\}$
              **if** [received $(c_p, r_p, estV_{c_p}, estL_{c_p})$ from $c_p$] **then**    $\{$*p received estV$_{c_p}$ from $c_p$*$\}$
                  $estV_p \leftarrow estV_{c_p}$
→                 $estL_p \leftarrow estL_{c_p}$
                  $ts_p \leftarrow r_p$
                  send $(p, r_p, ack)$ to $c_p$
              **else**    $\{$*p suspects that $c_p$ crashed*$\}$
                  send $(p, r_p, nack)$ to $c_p$

          **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*$\}$
              **if** $p = c_p$ **then**
                  **wait until** [for $\left\lceil \frac{(n+1)}{2} \right\rceil$ processes $q$ : received $(q, r_p, ack)$ **or** $(q, r_p, nack)$]
                  **if** [for $\left\lceil \frac{(n+1)}{2} \right\rceil$ processes $q$ : received $(q, r_p, ack)$] **then**
                      *R-broadcast* $(p, r_p, estV_p, estL_p, decide)$
                  **end if**
              **end if**
      **end while**

  **when** *R-deliver* $(q, r_q, estV_q, estL_q, decide)$    $\{$*if p R-delivers a decide message, p decides accordingly*$\}$
      **if** $state_p = undecided$ **then**
          ***decide***(*estV$_q$*)
→         $procList_p \leftarrow estL_q$    $\{$*updates the process list for the next execution*$\}$
          $state_p \leftarrow decided$
      **end if**
  **end when**
```

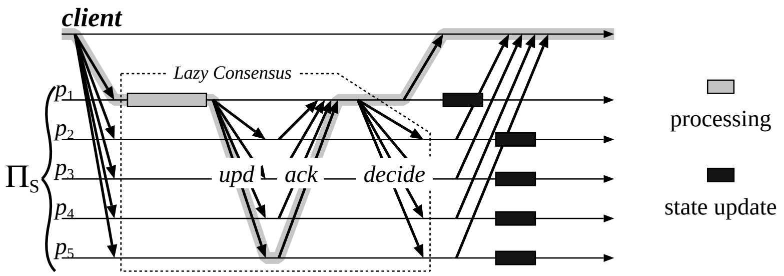

Figure 8: Semi-passive replication (good run). The critical path request-response is highlighted in gray. The execution of the Lazy Consensus is also depicted in Fig. 6.

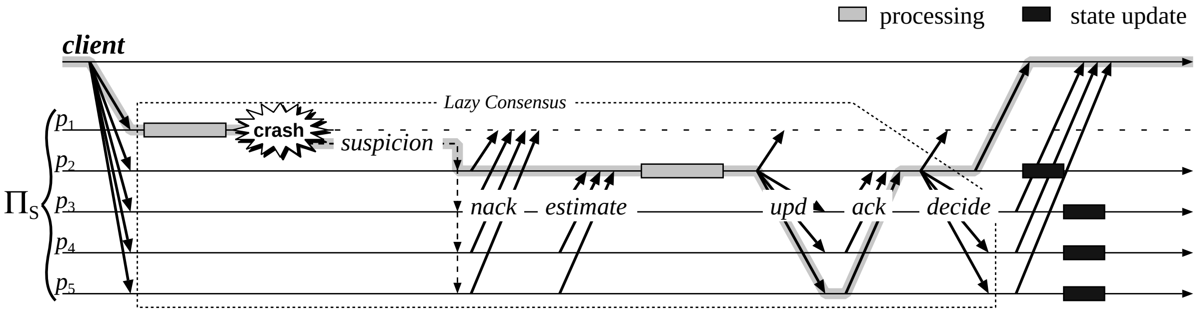

Figure 9: Semi-passive replication with one failure (worst case). The critical path request-response is highlighted in gray. The execution of the Lazy Consensus in the case of one crash is also depicted in Fig. 7.

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).

7.2 Semi-Passive Replication in the Case of One Crash

Figure 9 ilustrates 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. 8). 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 details, 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 acknowledgement 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. 9), and from that point on, the scenario is similar to the "good run" case of the previous section (compare with Fig. 8).

8 Conclusion

Semi-passive replication is a replication technique which 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.

The semi-passive replication algorithm proposed in this paper is based on solving the problem of Lazy Consensus. Lazy Consensus extends the usual definition of the Consensus problem with a property of Laziness, which is the key to the restrained use of resources in semi-passive replication. The semi-passive replication algorithm however only relies on the conventional properties of Consensus for its correctness. Even though we have not discussed this issue, other Consensus algorithms can easily be adapted to solve Lazy Consensus (e.g., [20, 29, 34, 35]).

As mentioned above, the main advantage of our semi-passive replication algorithm is that it does not rely on a group membership or on any other mechanism that forces the crash of processes. This may however give the wrong impression that housekeeping issues, such as eventually excluding crashed processes or adding new processes dynamically, are incompatible with semi-passive replication. This is not the case as discussed in [11, 12].

Other contributions can also be mentioned. First, we provide a specification framework that includes most of the existing replication techniques. Indeed, instead of overemphasizing the specificities of one given technique, we have tried to *harmonize* the specifications by stressing their common points rather than their difference. This clearly comes in contrast with previous work (e.g., [31, 30, 9]). Second, our semi-passive replication algorithm follows the exactly same protocol as active replication from the standpoint of the clients. This, combined with the fact that both replication techniques can be implemented based on consensus, makes it much easier for both techniques to coexist [16].

Acknowledgments

We would like to thank Fernando Pedone for his early feedback on the specification of replication techniques, and Péter Urbán for his enlightened suggestions that greatly simplified some of the proofs. In addition, we would like to express our gratitude to the following persons for their constructive critics and encouragements: Jean-Yves Le Boudec, Jean-Charles Fabre, Marc-Olivier Killijian, Dahlia Malkhi, Keith Marzullo, Friedemann Mattern, Hein Meling, Achour Mostefaoui, Stefan Pleisch, Ravi Prakash, Michel Raynal, Matthias Wiesmann.

References

[1] M. K. Aguilera, W. Chen, and S. Toueg. Quiescent reliable communication and quiescent consensus in partitionable networks. TR 97-1632, Dept. of Computer Science, Cornell University, Ithaca, NY, USA, June 1997.

[2] M. K. Aguilera, W. Chen, and S. Toueg. Failure detection and consensus in the crash-recovery model. In *Proceedings of the 12th International Symposium on Distributed Computing (DISC)*, volume 1499 of *Lecture Notes in Computer Science*, pages 231–245, Andros, Greece, September 1998. Springer-Verlag.

[3] M. K. Aguilera, W. Chen, and S. Toueg. Using the heartbeat failure detector for quiescent reliable communication and consensus in partitionable networks. *Theoretical Computer Science*, 220(1):3–30, June 1999. Special issue on distributed algorithms.

[4] 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.

[5] H. Attiya and J. Welch. *Distributed Computing*. Mc Graw Hill, 1998.

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

[7] K. P. Birman, T. Joseph, T. Raeuchle, and A. El Abbadi. Implementing fault-tolerant distributed objects. *IEEE Transactions on Software Engineering*, 11(6):502–508, June 1985.

[8] N. Budhiraja. *The Primary-Backup Approach: Lower and Upper Bounds*. PhD thesis, Dept. of Computer Science, Cornell University, Ithaca, NY, USA, June 1993. 93/1353.

[9] 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.

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

[11] B. Charron-Bost, X. Défago, and A. Schiper. Time vs space in fault-tolerant distributed systems. In *Proceedings of the 6th IEEE International Workshop on Object-oriented Real-time Dependable Systems (WORDS'01)*, Rome, Italy, January 2001.

[12] X. Défago, P. Felber, and A. Schiper. Optimization techniques for replicating CORBA objects. In *Proceedings of the 4th IEEE International Workshop on Object-oriented Real-time Dependable Systems (WORDS'99)*, pages 2–8, Santa Barbara, CA, USA, January 1999.

[13] X. Défago, A. Schiper, and N. Sergent. Semi-passive replication. In *Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 43–50, West Lafayette, IN, USA, October 1998.

[14] Xavier Défago. *Agreement-Related Problems: From Semi-Passive Replication to Totally Ordered Broadcast*. PhD thesis, École Polytechnique Fédérale de 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 *Second IFIP International Working Conference on Distributed Applications and Interoperable Systems (DAIS'99)*, pages 375–387, Helsinki, Finland, June 1999.

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

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

[19] V. Hadzilacos and S. Toueg. A modular approach to fault-tolerant broadcasts and related problems. TR 94-1425, Dept. of Computer Science, Cornell University, Ithaca, NY, USA, May 1994.

[20] M. Hurfin, R. Macêdo, M. Raynal, and F. Tronel. A general framework to solve agreement problems. In *Proceedings of the 18th Symposium on Reliable Distributed Systems (SRDS)*, pages 56–67, Lausanne, Switzerland, October 1999.

[21] M. Hurfin, A. Mostéfaoui, and M. Raynal. Consensus in asynchronous systems where processes can crash and recover. In *Proceedings of the 17th Symposium on Reliable Distributed Systems (SRDS)*, pages 280–286, West Lafayette, IN, USA, October 1998.

[22] L. Lamport. The implementation of reliable distributed multiprocess systems. *Computer Networks*, 2:95–114, 1978.

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

[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 *Proceedings of the 25rd International Symposium on Fault-Tolerant Computing (FTCS-25)*, pages 61–66, Pasadena, CA, USA, 1995.

[25] A. Mostéfaoui and M. Raynal. Solving consensus using Chandra-Toueg's unreliable failure detectors: A general quorum-based approach. In *Proceedings of the 13th International Symposium on Distributed Computing (DISC)*, number 1693 in Lecture Notes in Computer Science, pages 49–63, Bratislava, Slovak Republic, September 1999.

[26] R. Oliveira, R. Guerraoui, and A. Schiper. Consensus in the crash-recover model. Technical Report 97/239, École Polytechnique Fédérale de Lausanne, Switzerland, August 1997.

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

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

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

[30] F. Schneider. Replication management using the state-machine approach. In S. Mullender, editor, *Distributed Systems*, ACM Press Books, chapter 7, pages 169–198. Addison-Wesley, second edition, 1993.

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

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

[33] R. Van Renesse, K. P. Birman, and R. Cooper. The HORUS system. Technical report, Dept. of Computer Science, Cornell University, Ithaca, NY, USA, 1993.

[34] C. Yahata, J. Sakai, and M. Takizawa. Generalization of consensus protocols. In *Proc. of the 9th International Conference on Information Networking (ICOIN-9)*, pages 419–424, Osaka, Japan, 1994.

[35] C. Yahata and M. Takizawa. General protocols for consensus in distributed systems. In *Proc. of the 6th Int'l Conf. on Database and Expert Systems Applications (DEXA'95)*, number 978 in Lecture Notes in Computer Science, pages 227–236, London, UK, September 1995. Springer-Verlag.

[36] H. Zou and F. Jahanian. Real-time primary-backup replications with temporal consistency. In *Proceedings of the 18th International Conference on Distributed Computing Systems (ICDCS-18)*, pages 48–56, Amsterdam, The Netherlands, May 1998.

A Correctness of the Semi-Passive Replication Algorithm

We prove that our algorithm for semi-passive replication (Algorithm 1, page 9) satisfies the properties of the Generic Replication Problem given in Section 4.1. The proof assumes that (1) procedure *LazyConsensus* solves the Lazy Consensus problem according to the specification given in Section 5.2.1 (ignoring the laziness property [11]), and (2) at least one replica is correct. Solving Lazy Consensus is discussed in Section 6. In fact, Lazy Consensus solves Consensus, which is enough to prove the correctness of the algorithm as a Generic Replication algorithm.

LEMMA 5. *[Termination] If a correct client $c \in \Pi_C$ sends a request, it eventually receives a reply.*

PROOF. The proof is by contradiction. Assume that there is a reply for which a correct client in Π_C never receives a reply. Let c denote this client and req_c the request. Because 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 16, at least one replica s' removes the request $req_{h_{s'}}$ at the head of $recvQ_{s'}$ (l.7, l.19). 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. 5.1), 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 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 6. *[Total order] If there is an event $update(req)$ such that a replica executes $update(req)$ as its i^{th} update event, then all replicas that execute the i^{th} update event also execute $update(req)$ as their i^{th} event.*

PROOF. Let some replica execute $update(req)$ as the i^{th} update, i.e., the replica reaches line 18 of the algorithm with $k = i$ and executes $state \leftarrow update(upd_{req}, state)$. This means that the replica has decided on $(req, upd_{req}, -)$ at line 16. By the Agreement property of Lazy Consensus, every replica that decides for $k = i$, also decides on $(req, upd_{req}, -)$ at line 16. Therefore, every replica that executes line 18 with $k = i$ also executes $state \leftarrow update(upd_{req}, state)$. □

LEMMA 7. *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. □

[11] See Sect. 5.4, p. 9.

LEMMA 8. *For any request req, every replica executes $update(req)$ at most once.*

PROOF. Whenever a replica executes $update(req)$ (line 18), 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 9. *[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 7 and Lemma 8. □

LEMMA 10. *[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 11. *Algorithm 1 solves the generic replication problem (defined in Section 4.1).*

PROOF. Follows directly from Lemma 6 (total order), Lemma 9 (update integrity), Lemma 10 (response integrity), and Lemma 5 (termination). □

B Correctness of the Lazy Consensus Algorithm

Here, we prove the correctness of our Lazy Consensus algorithm (Algorithm 2, page 13). The algorithm solves the weak Lazy Consensus problem using the $\Diamond\mathcal{S}$ failure detector, with a majority of correct processes. Lemma 13–16 are adapted from the proofs of Chandra and Toueg [10] for the Consensus algorithm with $\Diamond\mathcal{S}$.

LEMMA 12. *No correct process remains blocked forever at one of the* wait *statements.*

PROOF. There are three *wait* statements to consider in Algorithm 2 (l.21, l.34, l.44). 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 $c = procList[1]$ does not wait in Phase 2 (l.18), 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 $(-, r, estV, -, -)$ to the current coordinator $c = procList[((r-1) \bmod n) + 1]$ (l.14). Since a majority of the processes are correct, at least $\lceil \frac{(n+1)}{2} \rceil$ 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 $\lceil \frac{(n+1)}{2} \rceil$ message of the type $(-, r, estV, -, -)$ in Phase 2.
2. c crashes.

In the first case, every correct process eventually receives $(c, r, estV_c, -)$ (l.34). 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.34). So every correct process sends a message of the type $(-, r, ack)$ or $(-, r, nack)$ to c in Phase 3 (resp. l.39, l.41). Since there are at least $\lceil \frac{(n+1)}{2} \rceil$ correct processes, c cannot block at the *wait* statement of Phase 4 (l.44). This shows that all correct processes complete round r—a contradiction that completes the proof of the lemma. □

LEMMA 13. *[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 $(-,-,-,-,decide)$ (l.50)). 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 12, 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., $q = procList[((r-1) \bmod n)+1]$). Since q is correct, then it eventually sends a message to all processes at the end of Phase 2 (l.31):

   - If round r is the first round, then q does not wait for any message, and sends $(q, r, estV_q, -)$ to all processes at the end of in Phase 2.

   - For round $r > 1$, then all correct processes send their estimates to q (l.14). In Phase 2, q receives $\left\lceil \frac{(n+1)}{2} \right\rceil$ such estimates, and sends $(q, r, estV_q, -)$ 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.34), eventually receives it, and replies with an *ack* to q (l.39). 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 $(-, r, ack)$ (and no messages of the type $(-, r, nack)$), and q R-broadcasts $(q, r, estV_q, -, decide)$ (l.46). By the validity and agreement properties of Reliable Broadcast, eventually all correct processes R-deliver q's message (l.50) and *decide* (l.52)—a contradiction.

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

LEMMA 14. *[Uniform integrity] Every process decides at most once.*

PROOF. Follows directly from Algorithm 2, where no process decides more than once. □

LEMMA 15. *[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 $(-,-,-,-,decide)$ (l.50). 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 $(-,-,ack)$ in Phase 4 (l.44). Let r be the smallest round number in which $\left\lceil \frac{(n+1)}{2} \right\rceil$ messages of the type $(-, r, ack)$ are sent to a coordinator in Phase 3 (l.39). Let c denote the coordinator of round r, that is, $c = procList[((r-1) \bmod n) + 1]$. 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 $estV_{c'}$ in Phase 2 of round r' (l.31), then $estV_{c'} = estV_c$.

The proof is by induction on the round number. The claim trivially holds for $r' = r$. Now assume that the claim holds for all $r', r \leq r' < k$. Let c_k be the coordinator of round k, that is, $c_k = procList[((k-1) \bmod n) + 1]$. We will show that the claim holds for $r' = k$, that is, if c_k sends $estV_{c_k}$ in Phase 2 of round k (l.31), then $estV_{c_k} = estV_c$.

From Algorithm 2 it is clear that if c_k sends $estV_{c_k}$ in Phase 2 of round k (l.31) then it must have received estimates from at least $\left\lceil \frac{(n+1)}{2} \right\rceil$ processes (l.21).[12] Thus, there is some process p such that (1) p sent a (p, r, ack) message to c in Phase 3 of round r (l.39), and (2) $(p, k, estV_p, -, ts_p)$ is in $msgs_{c_k}[k]$ in Phase 2 of round k (l.22). Since p sent (p, r, ack) to c in Phase 3 of round r (l.39), $ts_p = r$ at the end of Phase 3 of round r (l.38). Since ts_p is nondecreasing, $ts_p \geq r$ in Phase 1 of round k. Thus, in Phase 2 of round k, $(p, k, estV_p, -, ts_p)$ is in $msgs_{c_k}[k]$ with $ts_p \geq r$. It is easy to see that there is no message $(q, k, estV_q, -, ts_q)$ in $msgs_{c_k}[k]$ for which $ts_q \geq k$. Let t be the largest ts_q such that $(q, k, estV_q, -, ts_q)$ in $msgs_{c_k}[k]$. Thus, $r \leq t < k$.

[12] Note that $r < k$ hence round k is not the first round.

In Phase 2 of round k, c_k executes $estV_{c_k} \leftarrow estV_q$ where $(q, k, estV_q, -, t)$ is in $msgs_{c_k}[k]$ (l.27). From Algorithm 2, it is clear that q adopted $estV_q$ as its estimate in Phase 3 of round t (l.36). Thus, the coordinator of round t sent $estV_q$ to q in Phase 2 of round t (l.31). Since $r \leq t < k$, by the induction hypothesis, $estV_q = estV_c$. Thus, c_k sets $estV_{c_k} \leftarrow estV_c$ in Phase 2 of round k (l.27). 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 $(q, r_q, estV_q, -, decide)$, and thus decides $estV_q$. By the uniform integrity property of Reliable Broadcast and the algorithm, process q must have R-broadcast $(q, r_q, estV_q, -, decide)$ in Phase 4 of round r_q (l.46). From Algorithm 2, some process q must have received $\left\lceil \frac{(n+1)}{2} \right\rceil$ messages of the type $(-, r_q, ack)$ in Phase 4 of round r_q (l.45). By the definition of r, $r \leq r_q$. From the above claim, $estV_q = estV_c$. □

LEMMA 16. *[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 17. *Every process that terminates the algorithm considers the same value for the process list $procList$ after termination.*

PROOF. The proof is a trivial adaptation of Lemma 15 (uniform agreement) to $estL$. □

LEMMA 18. *Given a sequence of Lazy Consensus problems, processes begin every instance of the problem with the same process list.*

PROOF. The proof is by induction on the instance number k. If $k = 0$, then all processes initialize the process list using the lexicographical order of processes (l.2). If $k > 0$, then it follows from Lemma 17 that, if the process list is the same for all processes at the beginning of instance $k - 1$, then it is the same for all processes at the beginning of instance k. □

LEMMA 19. *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 19, 25, 27, and 36. 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 19, 25 (by hypothesis the function giv never returns $\perp$) and line 27 (the process selects a value explicitly different from $\perp$).

At line 36, a process sets its estimate to a value received from the coordinator. This value is sent by the coordinator c_p at line 31. Before reaching this line, c_p changed its own estimate $estV_{c_p}$ at one of the following lines: 19, 25, or 27. As shown above, $estV_{c_p}$ is never set to $\perp$ at these lines. □

LEMMA 20. *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 $estV_p \leftarrow \textbf{eval}\ giv$ (line 19 or 25). By line 17, p proposes a value only if p is the coordinator of the round (i.e., $p = c_p$). Let us consider line 19 and line 25 separately.

Line 19: The test at line 18 ensures that line 19 is executed only during the first round. Before executing line 19, $estV_p$ of the coordinator p is trivially equal to $\perp$ (initial value).

Line 25: The result directly follows from the test at line 24. □

LEMMA 21. *[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 19 and 25). 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 19, once $estV_p \neq \perp$, it remains different from $\perp$ forever. By Lemma 20, p proposes a value only if $estV_p = \perp$. A contradiction with the fact that p proposes a value twice. □

LEMMA 22. *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 20 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 34, 35, 36). Since p is not suspected by a majority of processes in Π_S (assumption), a majority of processes adopt p's estimate. By Lemma 19, 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 21, 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 24 returns false, and q does not call giv at line 25. A contradiction with the fact that q proposes a value in round r_q. □

COROLLARY 23. *[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 22. □

Lemma 22 is obviously not necessary to prove the weak laziness property defined in Section 5.2.1. However, as stated in Footnote 9 on page 8, it is interesting to show that our algorithm ensures a property stronger than weak laziness. The property is established by Lemma 22.

THEOREM 24. *Algorithm 2 solves the weak Lazy Consensus problem using $\Diamond\mathcal{S}$ in asynchronous systems with $f = \lfloor \frac{n}{2} \rfloor$.*

PROOF. Follows directly from Lemma 13 (termination), Lemma 14 (uniform integrity), Lemma 15 (agreement), Lemma 16 (validity), Lemma 21 (proposition integrity), and Lemma 23 (weak laziness). □