# Semi-Passive Replication in the Presence of Byzantine Faults [*]

**HariGovind V. Ramasamy  Adnan Agbaria  William H. Sanders**
University of Illinois at Urbana-Champaign
1308 W. Main Street, Urbana IL 61801, USA
{ramasamy, adnan, whs}@crhc.uiuc.edu

## Abstract

*Semi-passive replication is a variant of passive replication that does not rely on a group membership service. Défago et al. [4] defined the semi-passive replication concept in the crash fault model and described a semi-passive replication algorithm based on a lazy consensus algorithm. In this paper, we consider semi-passive replication and lazy consensus for a Byzantine fault model. We present lazy Byzantine consensus algorithms for two system models: 1) a system with synchronous communication and partially synchronous processing, and 2) an asynchronous system augmented with unreliable fault detectors for Byzantine faults. We prove that our algorithms provide safety and liveness. Our algorithms are optimal in good runs, having a latency degree of 2. We describe how our algorithms can be tuned to obtain the desired levels of fault resilience or efficiency in the presence of faults. We also present optimizations to improve the performance of the algorithms.*

## 1 Introduction

Replication of critical components is a widely used approach for achieving tolerance of faults. However, general approaches to replication are not simple, because of the need to maintain replica consistency. The replica consistency problem deals with ensuring that the internal states of all correct replicas in the system are consistent with each other. The techniques that have been proposed to solve this problem fall into two fundamental classes: active and passive. In active replication (also called the state machine approach), e.g., [13], all replicas start with the same initial state, and process client requests in the same order, thus maintaining identical internal states. In passive replication (also called the primary-backup approach), e.g., [1], only one primary replica processes the client requests and updates the other backup replicas. If the primary is faulty, one of the backup replicas is selected to be the new primary.

Both active and passive replication techniques have their advantages and disadvantages. While active replication maintains good performance in the presence of failures, it requires that all replicas process each

[*]This research has been supported by DARPA contract F30602-00-C-0172.

request and that operations on the replicas be deterministic. (*Determinism* means that the result of an operation depends only on a replica's initial state and the sequence of operations that the replica has completed so far.) Passive replication, on the other hand, requires less processing power than active replication and does not require determinism, but can suffer from poor performance (relative to active replication) in the presence of failures. This potential drawback of passive replication has been attributed to its reliance on a group membership service to detect failures and reselect the primary replica. To understand this, consider that a group membership service excludes the primary from the group whenever it is suspected to have crashed and selects a new primary. This reconfiguration of the group can be quite expensive, resulting in poor response times to the client. If a conservative failure detection mechanism is used, in order to avoid unnecessary reconfigurations, then when the primary does crash, the responsiveness will be poor.

Défago, Schiper, and Sergent defined the semi-passive replication concept in [4, 5, 3]. Semi-passive replication is a technique that retains the principal advantages of passive replication (reduced processing power and the no-determinism requirement) and removes the main disadvantage (reliance on a group membership service). An additional advantage of semi-passive replication is that the client is oblivious to which server replica is the primary replica. The client sends its request to all the replicas, and the failure of the primary (or any replica) is transparent to the client process. Hence, in semi-passive replication, there is no need for the client to re-issue its request. Défago et al. also presented an algorithm that implements semi-passive replication using a sequence of lazy consensus operations. Lazy consensus is similar to standard consensus except for one difference: in standard consensus, all processes start with an initial value, whereas in lazy consensus, a process obtains its initial value only when necessary. When there are no faults, only the primary will obtain its initial value. Since obtaining of an initial value is equivalent to a replica's processing of a request, the "laziness" of the consensus results in the "passiveness" of the replication technique.

Défago et al.'s definition of semi-passive replication and lazy consensus, and their algorithms were for the fault model in which processes can fail only by crashing. In this work, we consider semi-passive replication and lazy consensus for a Byzantine fault model.

The key observation that we used to provide semi-passive replication in the presence of Byzantine faults was that having only one primary replica (as in the crash fault model) is not possible, because a Byzantine primary process could faithfully send the reply to the client, and send the updates to the backups in time, but perform the wrong processing to obtain the reply and the updates. Our solution is based on having a *primary committee* that consists of $(t + 1)$ replicas (where $t$ is the maximum number of replicas that could be Byzantine-faulty) and uses authenticated message exchanges. When there are no faults, only the primary committee executes requests and updates backups. Such a solution involves much less processing power than Byzantine fault-tolerant active replication techniques such as the PBFT algorithm [2], in which all correct replicas execute the requests. However, a consequence of using more than one replica to execute the request (which is necessary to tolerate Byzantine faults) is that deterministic processing is necessary (as in all active replication techniques). We first present a solution that requires determinism, and then later discuss how we can do away with that requirement.

Our approach to providing a Byzantine fault-tolerant semi-passive replication algorithm is to use a sequence of *lazy Byzantine consensus* algorithms. The lazy Byzantine consensus algorithm requires only the primary committee to obtain values and reach a decision when there are no faults. When faults do occur, the primary committee is reselected, and the new primary committee tries to reach a decision.

We provide lazy Byzantine consensus algorithms for two system models: 1) the synchronous communication and partially synchronous processing model [4], and 2) the asynchronous system model augmented with unreliable fault detectors for Byzantine faults [10]. Our solutions require only the minimum number of processes needed to solve consensus under each of those models. However, the efficiency of the solution depends on the number of processes (above the minimum number required) and the policy for reselecting the primary committee under faults. We present a few examples to illustrate this fact. We also prove that our algorithms satisfy the properties of lazy Byzantine consensus.

## 2 Semi-Passive Replication in the Byzantine Fault Model

In this section, we define the concept of semi-passive replication in the Byzantine fault model. We also define the lazy Byzantine consensus concept, and present a Byzantine fault-tolerant semi-passive replication algorithm that is based on the lazy Byzantine consensus algorithm.

### 2.1 Application Model and Notations

The application model that we consider is the client-server model. The service $\mathcal{S}$ consists of a set of $n$ server replicas $\{p_1, \cdots, p_n\}$ that are deterministic and initialized to the same internal state. A maximum of $t$ among the $n$ server replicas can be Byzantine-faulty. A $k$-subset of $\mathcal{S}$ consists of exactly $k$ distinct server replicas. We use $\mathcal{S}^k$ to denote the set containing all possible $k$-subsets of $\mathcal{S}$. $\mathcal{PC}$ denotes the set containing all the $(t+1)$-subsets of $\mathcal{S}$ that could constitute primary committees. $\mathcal{PC} \subseteq \mathcal{S}^{t+1}$.

For simplicity, we assume that there is a single, non-faulty client denoted by $c$. Communication between $c$ and the individual server replicas is asynchronous and takes place through point-to-point, FIFO, and reliable communication channels. Conceptually, we can consider $c$ as a serializing mechanism that accepts requests from multiple clients, enforces a total order among the client requests, and forwards the requests to all the server replicas. It is easy to see the similarity between $c$ and a real-world corporate gateway.

We use $req_k$ to denote the $k^{th}$ request from $c$ at any server replica. $req_k$ will be the same at all replicas (by our assumption of a single, non-faulty client). We use $resp_k^i$ to denote the response from $p_i$ for $req_k$. The response will be signed by $p_i$. The client $c$ *accepts* a response $r$ if it has received the identical response $r$ from $t+1$ replicas. We use $upd_k^i$ to denote the update in the internal state of $p_i$ as a consequence of processing the request $req_k$.

### 2.2 Specification of Byzantine Fault-Tolerant Semi-Passive Replication

We now define semi-passive replication in the Byzantine fault model by the following properties:

**Termination** If a correct client sends a request, it eventually *accepts* a response.

**Total Order** For any two *correct* replicas $p_i$ and $p_j$, the $k^{th}$ updates to their internal states, $upd_k^i$ and $upd_k^j$ respectively, are the same.

**Update Integrity** A *correct* replica $p_i$ executes $upd_k^i$ at most once, and only if the client sent $req_k$.

**Response Integrity** For any response $resp_k$ corresponding to the request $req_k$ *accepted* by the client, $upd_k^i$ is executed by some correct replica $p_i$.

**Weak Byzantine Parsimony** For any two elements $\mathcal{P}$ and $\mathcal{Q}$ of $\mathcal{PC}$ and for two correct replicas $p$ and $q$ such that $p \in \mathcal{P} \wedge p \notin \mathcal{Q}$ and $q \in \mathcal{Q} \wedge q \notin \mathcal{P}$, if both $p$ and $q$ process the same request $req_k$, then at least one of the following is true:

- At least one replica in $\mathcal{P}$ is suspected by some other replica in $\mathcal{P}$.
- At least one replica in $\mathcal{Q}$ is suspected by some other replica in $\mathcal{Q}$.

The definitions of the Termination, Total Order, Update Integrity, and Response Integrity properties have some words in italics to highlight the differences from the corresponding properties for the crash fault-tolerant case given in [3]. The differences are a consequence of the Byzantine fault model, in which faulty processes may behave arbitrarily. We define a new Weak Parsimony property[1] for a semi-passive replication system subject to Byzantine faults. The Parsimony property is used to specify the characteristic that distinguishes passive replication from active replication: reduced processing requirements. Normally, in the crash fault model, only one replica processes the requests. Our specification for the Byzantine fault model states that under normal circumstances, only the $(t + 1)$ replicas in some element of $\mathcal{PC}$ process the requests.

## 2.3 Lazy Consensus and Lazy Byzantine Consensus

In [3], the semi-passive replication algorithm is expressed as a sequence of (crash fault-tolerant) *lazy consensus* operations. Lazy consensus is a generalization of standard consensus. In the standard consensus algorithm, each process starts with an initial value. When the semi-passive replication algorithm is expressed as a sequence of lazy consensus algorithms, the decision to be reached by the consensus algorithm is the content of the update message that is generated after a request is processed. The contents of the update message give the modification to the internal state of a replica as a result of processing the request. If each process started with an initial value, then each of them should process the request. However, in that case, the replication algorithm would no longer qualify as "parsimonous." Hence, the computation of an initial value is deferred until necessary (hence the name *lazy* consensus). In the crash-fault-tolerant lazy consensus, this means that only the coordinator/primary process computes the initial value in the normal case (i.e., when

[1]The Weak Parsimony property in the crash fault model [3] states that "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."

there are no faults). Only when the coordinator is faulty (or suspected to be faulty by enough other processes) does another process become the coordinator and try to complete the consensus.

We follow a similar approach and express the semi-passive replication algorithm as a series of Byzantine fault-tolerant lazy Consensus (henceforth called *lazy Byzantine consensus* or LBC) algorithms. However, instead of having a primary process, we have a primary committee consisting of $(t+1)$ processes. $(t+1)$ is the minimum strength of the primary committee required to avoid the case in which the committee consists of all Byzantine replicas that faithfully send the required messages to other processes but carry out the wrong processing, resulting in wrong responses to client requests and bad updates to other server replicas. In lazy Byzantine consensus, only the primary committee computes the initial value in the normal case. When a primary committee member suspects another committee member, a new primary committee will be selected. The new committee members compute the initial value (if they haven't done so already) and try to complete the consensus. The reselection of the primary committee is the responsibility of the lazy consensus algorithm.

**Specification of Lazy Byzantine Consensus** "Proposing a value" from the point of view of lazy Byzantine consensus is equivalent to "processing a request" from the point of view of semi-passive replication. The lazy Byzantine consensus problem defined on the set of server replicas $\mathcal{S}$ is specified by the following properties:

**Termination** Every correct process eventually decides on some value.

**Uniform Integrity** Every *correct* process decides at most once.

**Agreement** No two correct processes decide differently.

**Uniform Validity** If a *correct* process decides on $v$, then $v$ was proposed by some *correct* process of $\mathcal{S}$.

**Propositional Integrity** Every *correct* process proposes a value at most once.

**Weak Byzantine Laziness** For any two elements $\mathcal{P}$ and $\mathcal{Q}$ of $\mathcal{PC}$ and for two correct processes $p$ and $q$ such that $p \in \mathcal{P} \wedge p \notin \mathcal{Q}$ and $q \in \mathcal{Q} \wedge q \notin \mathcal{P}$, if both $p$ and $q$ propose a value, then at least one of the following is true:

- At least one process in $\mathcal{P}$ is suspected by some other process in $\mathcal{P}$.
- At least one process in $\mathcal{Q}$ is suspected by some other process in $\mathcal{Q}$.

As before, the words in italics highlight the differences from the corresponding properties for the crash fault-tolerant lazy consensus given in [3]. The first five properties are known properties for any solution to the standard Consensus problem. We define a new Weak Byzantine Laziness property to ensure the Weak Byzantine Parsimony property of semi-passive replication. Note that it is possible to define stronger versions of the laziness property, but satisfying those properties would require a system in which failures are always correctly detected. With the weak Byzantine Laziness property, we allow for incorrect suspicions that may lead to the processes in two elements of $\mathcal{PC}$ to propose a value.

## 2.4 The Semi-Passive Replication Algorithm

Our Byzantine fault-tolerant semi-passive replication algorithm is simple because it gives only the client-server interaction, and delegates all the complexity in ensuring replica consistency to a lazy Byzantine consensus algorithm. We present two LBC algorithms later in Sections 3 and 4. The semi-passive replication algorithm is identical to the one presented in [3]; the only difference is that an LBC algorithm is invoked instead of a crash-fault-tolerant lazy consensus algorithm. To make this paper self-contained, we briefly describe the semi-passive replication algorithm here.

The semi-passive replication algorithm is executed by every server replica. Each replica maintains its own *receive queue* that contains the requests received from the clients and a *handled set* that contains the requests that have been processed. Any new client request (i.e., a request that is not already in the *receive queue* or in the *handled set*) is appended to the *receive queue*. A new instance of the lazy Byzantine consensus algorithm will be started by an invocation of a function *LazyByzConsensus(giv)* whenever the preceding instance has terminated and the *receive queue* is not empty. The argument *giv* (whose name is short for *g*et *i*nitial *v*alue) is a function that computes the initial value and returns it. Since obtaining an initial value (at the LBC level) is equivalent to processing the request (at the replication algorithm level), in order to satisfy the parsimonous property of semi-passive replication, the *giv* function must be invoked only by current members of the primary committee during the execution of the LBC algorithm. The function *giv* selects the client request at the head of the *receive queue*, processes the request, and returns the initial value for the Consensus. Since the client requests are received in the same order at all processes, all correct processes must have the same initial value. The initial value is a 3-tuple containing 1) the selected client request, 2) the update that should be applied to the replica state once a decision has been reached, and 3) the response that should be sent to the client. When the consensus terminates, based on its decision, each server replica sends its response to the client, updates its local state, removes the handled request from the *receive queue*, and adds the request to the *handled set*. The responses sent to the client are signed by the sending server replicas. The client waits for identical responses from $(t + 1)$ server replicas before it accepts the response.

We need to emphasize that our semi-passive replication algorithm relies only on the Laziness property of the lazy Byzantine consensus algorithm to satisfy the Parsimony property of semi-passive replication. Substituting our lazy Byzantine consensus algorithm with any of the known standard Consensus algorithms (e.g., [2, 6, 10, 11]) will not compromise replica consistency, but might lead to a violation of the Parsimony property, so that the replication algorithm would no longer be “semi-passive.”

### 2.5 Solving the Lazy Byzantine Consensus Problem

As mentioned before, the LBC problem is a generalization of the standard Byzantine Consensus problem. Hence, the FLP impossibility result [7] applies to the LBC problem. This has the implication that no deterministic algorithm can solve LBC in an asynchronous system in the presence of a single faulty process. To circumvent this impossibility result, we take the common approach of strengthening the timing assumptions of the base asynchronous system model. In particular, we consider two system models: 1) the partial synchrony model of Dwork et al. [6], and 2) the asynchronous system model augmented with unreliable fault detectors for Byzantine faults [10]. We present a solution to the LBC problem in each of these system models next in Sections 3 and 4.

## 3 Lazy Byzantine Consensus in the Partially Synchronous Processing and Synchronous Communication Model

In this section we present an algorithm for lazy Byzantine consensus (LBC) as defined in Section 2.3. The algorithm, which we call the *LBC-Psync algorithm*, is based on the partial synchrony model of Dwork et al. given in [6]. We briefly describe the model below.

### 3.1 System Model

The system of $n$ server processes, $\mathcal{S} = \{p_1, \cdots, p_n\}$, follows the partial synchrony model of synchronous communication and partially synchronous processing [6]. The server processes are connected by a synchronous network. Since we assume synchronous communication, there is a fixed upper bound $\Delta$ on the time it takes it takes messages to be delivered. $\Delta$ is known by every process. In particular, no messages are lost. In addition, if we assume synchronous processing, there is an upper bound $\Phi$, which is known by every process, on the rate at which one processor's clock can run faster than another's. However, since we assume partially synchronous processing, there is a *global stabilization time* (GST), unknown to the processes, such that the processing respects the upper bound $\Phi$ from time GST onward.

A *correct* server process behaves according to its specification; a *faulty* process doesn't. Up to $t$ processes may be corrupted by an adversary and might behave arbitrarily. We assume *authenticated communication*, in which messages can be signed with the name of the sender process in such a way that the signature cannot be forged by any other process. Hence, our fault model is the Authenticated Byzantine fault model. In it, the minimum number of processes required to solve consensus is $n > 2t$ [6]. Hence, we require that $t < \frac{n}{2}$.

As mentioned in [6], any algorithm that solves Byzantine consensus in the GST model is required to satisfy the safety conditions, even if $\Phi$ does not hold eventually. On the other hand, the algorithm needs to satisfy termination only in case $\Phi$ holds eventually.

## 3.2 Overview

The key idea in our algorithm is that normally only the $t + 1$ processes that constitute the primary committee (denoted by $pc$) obtain the initial values and send them to all processes. When any process obtains identical initial values from $(t + 1)$ processes, then it can decide on that value, since at least one of the sender processes must be correct and a correct $pc$ member always sends the correct initial value. However, if one or more of the $pc$ members are faulty, then the faulty $pc$ members may not send their initial values in a timely manner, may not send them at all, or may send the wrong initial value. In such a case, the $pc$ will be reselected. Primary committee reselection can be initiated only by a process that is currently a $pc$ member. Since at least one process in the $pc$ is correct, if a decision is not reachable, $pc$ reselection will occur. After reselection, the new $pc$ members will try to reach consensus and repeat the steps outlined above.

A correct $pc$ member will initiate reselection if 1) it has waited what it thinks is "long enough" without receiving the initial values of all the other $pc$ members, 2) the initial value of another $pc$ member differs from this $pc$ member's own initial value (which is not allowed since processing is deterministic, and the client requests are serialized in the same order at all processes), or 3) it finds that a fellow $pc$ member has already reselected a new $pc$.

A *reselection policy* determines what the next $pc$ will be. In other words, the policy defines the elements in the set $\mathcal{PC}$ and the ordering among them. The policy must be deterministic and identical at all processes. We make the following assumption about the reselection policy:

**Assumption 1** *In an infinitely long run consisting of infinitely many $pc$ reselection rounds, where at most $t$ out of $n$ processes in $\mathcal{S}$ are faulty and $n > 2t$, a $(t + 1)$-subset of $\mathcal{S}$ consisting entirely of correct processes must become the $pc$ infinitely often.*

Assumption 1 has two consequences:

1. The set $\mathcal{PC}$ (defined in Section 2.3) should contain at least one $(t+1)$-subset of $\mathcal{S}$ that consists entirely of correct processes, irrespective of which processes of $\mathcal{S}$ are faulty, as long as no more than $t$ of them are faulty.

2. Each element of $\mathcal{PC}$ should be selected as the $pc$ infinitely often in a infinitely long run consisting of infinitely many $pc$ reselection rounds.

Later, in Section 3.8, we give some examples of reselection policies that satisfy the above assumption. Intuitively, we expect that at some point after the GST, if the $pc$ consists only of correct members, then a decision will be reached.

## 3.3 Notations and Message Types

Each process $p_i$ maintains the following data structures:

- $V_i$ is an $n$-vector for storing the values proposed by different processes. Specifically, $V_i[i]$ contains $p_i$'s estimate of the decision value. Initially, $V_i[j] = \perp$ for all $j$.
- $cn_i$ denotes $p_i$'s current committee number (which indicates the $pc$ reselection round number). This is initialized to value $0$.
- $pc_i$ denotes $p_i$'s perception of the primary committee.
- $decided_i$ indicates whether $p_i$ has decided. Initially, all processes are undecided and $decided_i$ is *false*.

The primary committee corresponding to the committee number $cn_i$ is obtained by a call to a function $committee()$ with $cn_i$ as the argument. $committee(x)$ for a positive integer $x$ is a deterministic function that gives the $pc$ for the $x^{th}$ round of primary reselection. The reselection policy determines the implementation of *committee(x)*.

We use $\langle M \rangle_{p_i}$ to denote a message $M$ signed by process $p_i$. Our algorithm uses three message types:

***propose*:** A $pc$ member $p_i$ conveys its initial value for consensus to other processes by means of a *propose* message. The message is of the form $\langle \text{PROPOSE}, v \rangle_{p_i}$, where $v$ indicates the proposed initial value.

***decision*:** When a process has received $t + 1$ *propose* messages (possibly including one from itself) that have identical initial values, it takes this value as its own decision value. It then sends a *decision* message to all the other processes. The message carries the $t + 1$ signed proposed values as proof. The message is of the form $\langle \text{DECISION}, v, \text{proof} \rangle_{p_i}$, where $v$ is the decision value and "proof" is the set of signed *propose* messages from $t + 1$ processes.

***reselect*:** A $pc$ member may initiate reselection of the $pc$ (for any of the three reasons stated above in Section 3.2) by sending a *reselect* message to all the processes. The valid *reselect* message from a process $p_i$ is of the form $\langle \text{RESELECT}, cn_i, \text{proof} \rangle_{p_i}$, where $committee(cn_i)$ is $p_i$'s newly selected primary committee and $(p_i \in committee(cn_i)) \vee (p_i \in committee(cn_i\text{-}1))$. The proof field is null if $p_i \in committee(cn_i\text{-}1)$; otherwise, the field should contain a valid *reselect* message sent by some $p_j \in committee(cn_i\text{-}1)$. That enforces a property of the algorithm, namely that $pc$ reselection can be initiated only by a current $pc$ member and should involve the participation of only the current $pc$ members and the next $pc$ members.

### 3.4 Detailed Description of the Algorithm

We now present the full algorithm that solves lazy Byzantine consensus in the partial synchrony model described above, provided that at least $\lfloor \frac{n+1}{2} \rfloor$ of the processes are correct and Assumption 1 holds. The pseudocode for the full algorithm executed by a process $p_i$ is given in Figure 1. We omit details of how we check the proper format of each received message. From the point of view of the semi-passive replication algorithm, the algorithm in Figure 1 represents the computations at $p_i$ and the communication that takes place between the server replica $p_i$ and other server replicas in order to service a single client request. In

```
Block 1: Initialization
 $V_i[j] \leftarrow \perp$, for all $0 \leq j < n$
 $cn_i \leftarrow 0,\;\; pc_i \leftarrow committee(cn_i),\;\; decided_i \leftarrow false$

Block 2: A $pc$ member obtains its initial value and proposes the value to all
function propose-now()
  $V_i[i] \leftarrow giv()$
  send $\langle$PROPOSE, $V_i[i]\rangle_{p_i}$ to all
  scheduleTimeout()

Block 3: $p_i$ has got propose messages from $(t+1)$ processes with identical value $v$ and decides on $v$
function decide-now()
  $decide(v),\;\; decided_i \leftarrow true$
  send $\langle$DECISION, $v$, proof$\rangle_{p_i}$ to all
  if $p_i \in pc_i$ then,  cancelTimeout()

Block 4: $pc$ member $p_i$ moves to the next $pc$ reselection round
function reselect-now()
  cancelTimeout()
  $cn_i \leftarrow cn_i + 1,\;\; pc_i \leftarrow committee(cn_i)$
  send $\langle$RESELECT, $cn_i$, proof$\rangle_{p_i}$ to all, with proof $=$ null
  if $[(p_i \in pc_i) \wedge (V_i[i] \neq \perp) \wedge (decided_i = false)]$ then,  scheduleTimeout()

Block 5: $pc$ member $p_i$ has not proposed or decided yet; hence it proposes
when $[(decided_i = false) \wedge (p_i \in pc_i) \wedge (V_i[i] = \perp)]$
  propose-now()

Block 6: $p_i$ has not decided yet and receives a propose message from $p_j$
when [$(decided_i = false) \wedge$ (received $\langle$PROPOSE, $v\rangle_{p_j}$ from $p_j$)]
  if $V_i[j] = \perp$ then
    $V_i[j] \leftarrow v$
    if $p_i \in pc_i$ then
      if $V_i[i] = \perp$ then,  propose-now()
      if $[(V_i[i] \neq V_i[j]) \wedge (p_j \in pc_i)]$ then,  reselect-now()
    if [at least (t+1) non-null elements of vector $V_i$ have identical value $v$] then
      decide-now()

Block 7: $p_i$ has not decided yet and receives a decision message from $p_j$
when [$(decided_i = false) \wedge$ (received $\langle$DECISION, $v$, proof$\rangle_{p_j}$ from $p_j$ with valid proof)]
  decide-now()

Block 8: $p_i$ receives a reselect message for the $pc$ reselection round number $cn_i + 1$
when received $\langle$RESELECT, $cn_j$, proof$\rangle_{p_j}$ for $pc$ reselection round $cn_j = cn_i + 1$
            from $(p_j \in committee(cn_j\text{-}1)) \vee (p_j \in committee(cn_j))$ with valid proof
  if $p_i \in pc_i$ then,  reselect-now()
  else
    $cn_i \leftarrow cn_i + 1,\;\; pc_i \leftarrow committee(cn_i)$
    if $p_i \in pc_i$ then
      send $\langle$RESELECT, $cn_i$, proof$\rangle_{p_i}$ to all, with proof $=$ reselect message received from $p_j$
      if $[(V_i[i] \neq \perp) \wedge (decided_i = false)]$ then,  scheduleTimeout()
    else,  forward the reselect message to all

Block 9: $pc$ member $p_i$ has proposed but other $pc$ members have not sent their propose messages in time
when [timeout $\wedge$ $(p_i \in pc_i)$],  reselect-now()
```

**Figure 1. The Lazy Byzantine Consensus Algorithm at a Process $p_i$**

other words, given that the semi-passive replication algorithm is expressed as a sequence of lazy Byzantine consensus problems, the following presents the details of how the $k^{th}$ instance of the problem (corresponding to the $k^{th}$ client request received) is solved.

A **when** block of statements is enabled as soon as the guard condition becomes true. The enabled **when** blocks are executed in the order in which they become enabled. We assume that the execution of a **when** block is atomic and not interleaved with the execution of another **when** block (or another instance of the same **when** block).

The *propose-now()* function (Block 2) obtains the initial value for the consensus algorithm by invoking the *giv()* function. (This call to the *giv* function is equivalent, from the semi-passive replication algorithm's perspective, to the processing of a client request.) The *giv()* function computes a non-null initial value and returns it. The obtained initial value is sent in a *propose* message to all the processes. Only $pc$ members that have the null value $\perp$ as their initial value and have not yet decided will invoke the *propose-now()* function (from Block 5 and Block 6). To ensure liveness, a $pc$ member that has sent its initial value schedules a timeout by which it expects other $pc$ members to send their respective initial values.

The *decide-now()* function (Block 3) is invoked by any process $p_i$ that has received at least $t + 1$ *propose* messages with identical values (say $v$) either individually or collectively as the proof part of a *decision* message. $p_i$ takes $v$ as its decision value (by executing $decide(v)$), updates $decided_i$ to the *true* value, and sends its own *decision* message with correct proof to all. Additionally, if $p_i$ is a $pc$ member, it cancels the timeout it scheduled when it sent out its own *propose* message or when it switched to the current committee number (whichever occurred last).

The *reselect-now()* function (Block 4) is invoked by a current $pc$ member $p_i$ to initiate the next $pc$ reselection round. Since the purpose of a scheduled timeout is to initiate the next $pc$ reselection round if initial values have not been received in time from other $pc$ members, a timeout previously scheduled but hasn't yet expired is now unneeded, and is therefore cancelled. The function increments the committee number $cn_i$, updates the primary committee $pc_i$, and sends a *reselect* message with the new committee number to all the processes. Since $p_i$ was a $pc$ member when invoking the function, the *reselect* message does not need any proof. Additionally, if $p_i$ is a member of the new $pc$ as well, and has already sent its initial value, it schedules a timeout by which it expects to receive the initial values from the members of the new $pc$.

In Block 5, $p_i$ invokes the *propose-now()* function if it is a $pc$ member, and has not yet decided or proposed.

Block 6 describes the actions taken when $p_i$ has not yet decided and receives a *propose* message from process $p_j$ for the first time. $p_i$ updates the $j^{th}$ element in the $V_i$ vector. If $p_i$ is a $pc$ member and has not yet sent its *propose* message, it does so now; it then checks whether its initial value and $p_j$'s initial value are the same. If the values are different (indicating that $p_j$ is faulty) and $p_j$ is currently a $pc$ member, $p_i$ initiates the next $pc$ reselection round by calling the function *reselect-now()*. The check to see whether $p_j$ is a $pc$ member prevents the situation in which a malicious $p_j$ that is not currently a $pc$ member sends a wrong initial value to initiate $pc$ reselection when all the $pc$ members are correct. Finally, $p_i$ checks whether it has obtained $(t + 1)$ *propose* messages with identical values; if it has, it invokes the *decide-now* function.

In Block 7, if $p_i$ receives a valid *decision* message with valid proof in the form of $(t+1)$ *propose* messages with identical values of $v$, then $p_i$ invokes the *decide-now()* function.

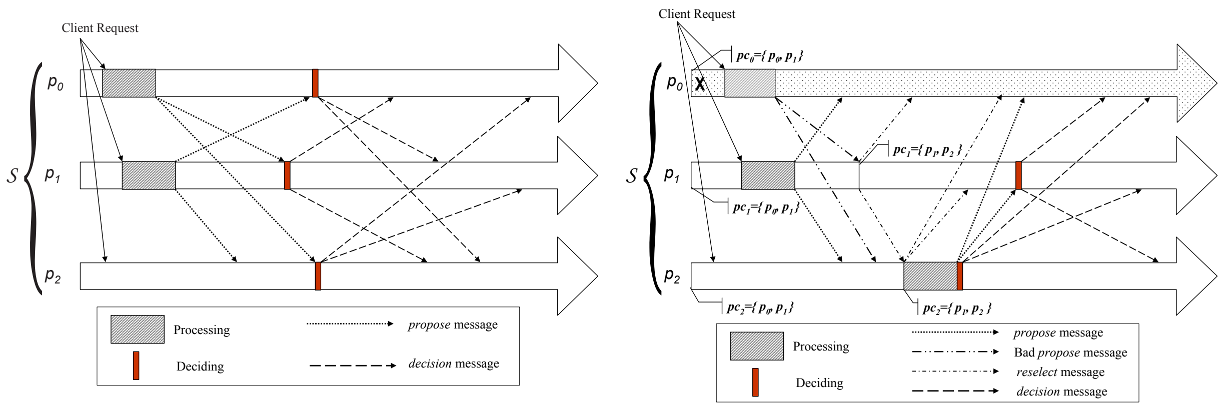
As described in Section 3.3, a *reselect* message for a new $pc$ reselection round $cn_j$ can be sent by process $p_j$ only if it is a member of $committee(cn_j)$, a member of $committee(cn_j\text{-}1)$, or both. If $p_j$ is a member of $committee(cn_j)$ but not a member of $committee(cn_j\text{-}1)$, the *reselect* message must carry a valid proof in the form of another *reselect* message sent by some process $p_k$ that is a member of $committee(cn_j\text{-}1)$. Block 8 describes $p_i$'s reaction to a valid *reselect* message for a new $pc$ reselection round $cn_i + 1$ from process $p_j$. If $p_i$ is currently a $pc$ member, it invokes the *reselect-now()* function, which increments $cn_i$, updates $pc_i$, and sends its own *reselect* message for that reselection round. If $p_i$ is not currently a $pc$ member, it increments $cn_i$, updates $pc_i$, and checks to see whether it is a member of the $pc$ corresponding to the new reselection round. If it is a member of the new $pc$, then $p_i$ sends a *reselect* message with $p_j$'s *reselect* message as proof. Additionally, if $p_i$ has already sent its *propose* message (in some previous reselection round in which it was a $pc$ member), it schedules a timeout by which it expects to hear from other members of the new $pc$ about their initial values. If $p_i$ is a member of neither the old $pc$ nor the new $pc$, then $p_i$ immediately relays $p_j$'s *reselect* message to all the processes. This forwarding of $p_j$'s reselect message to all the processes by $p_i$ prevents situations in which $p_j$ is a malicious $pc$ member sending a *reselect* message for a new $pc$ reselection round to only a subset of correct processes. The forwarding is not necessary when $p_i$ is a $pc$ member or a member of $committee(cn_j)$, because in those cases, $p_i$ will have to send its own *reselect* message.

Block 9 shows that if $p_i$ is a $pc$ member and has waited long enough but still hasn't been able to decide on a value (because $(t+1)$ *propose* messages with identical values have not yet been received), then $p_i$ initiates the next $pc$ reselection round.

### 3.5 Example Scenarios

Figure 2(a) presents an execution of the LBC-Psync algorithm when a client request is received at three server processes $p_0$, $p_1$, and $p_2$. The primary committee consists of $p_0$ and $p_1$. $p_0$ and $p_1$ process the request to obtain the same initial value which they send in a *propose* message to all the processes. Once a process receives *propose* messages from both the $pc$ members with identical values, it decides on that value, and sends a *decision* message to all the processes.

Figure 2(b) presents another execution of the LBC-Psync algorithm when a client request is received at three server processes, but in this case, one of the $pc$ members, namely $p_0$, is Byzantine-faulty. Both the $pc$ members, $p_0$ and $p_1$ process the request to obtain their initial values, but $p_0$ sends the wrong value in its *propose* message to all the processes. When $p_1$ receives this message, it initiates the next round of $pc$ reselection. The reselection policy is such that (in this example) $p_1$ and $p_2$ become the $pc$ members in that round. After becoming a $pc$ member, $p_2$ processes the request and sends out its *propose* message. Since $p_2$ had already received $p_1$'s *propose* message with initial value identical to its own initial value, it decides on



(a) Normal Case: No Faults and $\Phi$ Holds (b) Malicious Behavior of $p_0$

**Figure 2. Example Scenarios for the LBC-Psync Algorithm**

that value, and sends out a *decision* message. Similarly, when $p_1$ receives $p_2$'s *propose* message, it decides, and send out a *decision* message.

## 3.6 Safety and Liveness of the LBC-Psync algorithm

In this section, we will sketch the proofs that the LBC-Psync algorithm satisfies safety and liveness properties. First, we observe from Figure 1 that there are only three places in the algorithm where a correct $pc$ member will invoke the *reselect-now()* function to switch to the next $pc$ reselection round and send a *reselect* message to all the processes. This observation could be stated as follows:

**Observation 1** *There are only three cases in which a correct $pc$ member $p_i$ will initiate the next $pc$ reselection round. They are (1) a timeout occurs (Block 9), which means that $p_i$ has waited what it thinks is "long enough" and has not yet received the initial value of the other $pc$ members, (2) the initial value of some fellow $pc$ member differs from its own initial value (Block 6), or (3) it finds that a fellow $pc$ member has already reselected a new $pc$ (Block 8).*

**Lemma 3.1:** After GST, if the $pc$ consists entirely of correct processes, then no further $pc$ reselection rounds will occur.

**Proof (Sketch):** We prove this lemma by showing that the three situations in Observation 1 are not possible when, sometime after GST, the $pc$ consists entirely of correct processes.

Case (1) will not occur since all the members will (by the algorithm specification) send their initial values through *propose messages* if they have not already done so and because all $pc$ members will receive the initial values of other members in a timely manner, since $\Phi$ holds.

Case (2) will not occur because processing is deterministic, and the client requests are serialized in the same order at all processes. Hence, the initial values of all correct processes will be the same.

Because case (1) and case (2) are not possible and since all other $pc$ members are correct, a $pc$ member will have no reason to initiate a new $pc$ reselection round. Hence, a correct $pc$ member will not receive a *reselect* message from another $pc$ member. Thus, case (3) is also not possible. Hence, after GST, if the $pc$ consists entirely of correct processes, no further $pc$ reselection will occur. $\square$

**Lemma 3.2:** If GST exists, then the number of $pc$ reselection rounds after GST is finite.

**Proof (Sketch):** The reselection policy (by Assumption 1) guarantees that there is at least one $(t+1)$-subset consisting entirely of correct processes (say $P$) that is an element of $\mathcal{PC}$. The policy also guarantees that in a run of infinitely many reselection rounds, $P$ will become the primary committee infinitely often. Hence, if GST exists, $P$ must become the primary committee some time after GST. From Lemma 3.1, it follows that once $P$ becomes the primary committee after GST, then there will be no more reselection rounds. Hence, the number of $pc$ reselection rounds after GST is finite. $\square$

**Theorem 3.3:** Every correct process eventually decides on some value. *(Termination)*

**Proof (Sketch):** It is easy to see that if any correct process decides, it sends a *decision* message to all the processes. Since messages are not lost, eventually all correct processes will receive this message and also decide.

Now let us consider the case in which no correct process decides. Since the $pc$ consists of $(t+1)$ members, it must have at least one correct member. This correct member will timeout (Block 9) and initiate the next $pc$ reselection round. By Lemma 3.1 and Lemma 3.2, it follows that the latest time when any correct process will have to wait until before deciding would be the time when an all-correct-$(t+1)$-subset becomes the $pc$ after GST. Hence, termination is satisfied. $\square$

**Theorem 3.4:** Every correct process decides at most once. *(Uniform Integrity)*

**Proof (Sketch):** When a correct process $p_i$ decides on a value $v$, it updates $decided_i$ to $true$. The only place in our algorithm where this update is done is in the *decide-now()* function (Block 3). The function is invoked at only two places in the algorithm: Block 6 and Block 7. Those blocks are executed only when the value of $decided_i$ is $false$. Hence every correct process decides at most once. $\square$

**Theorem 3.5:** If a correct process decides on $v$, then $v$ was proposed by some correct process of $\mathcal{S}$. *(Uniform Validity)*

**Proof (Sketch):** A correct process $p_i$ decides on a value $v$ by calling the *decide-now()* function (Block 3). The function is invoked only after $p_i$ has received at least $t+1$ valid *propose* messages with identical values (say $v$) either individually (Block 6) or collectively as the proof part of a *decision* message (Block 7). In both cases, at least one of the $t+1$ *propose* messages must be from a correct process. Thus, Uniform Validity is satisfied. $\square$

**Theorem 3.6:** No two correct processes decide differently. *(Agreement)*

**Proof (Sketch):** Let us assume that two correct processes $p_i$ and $p_j$ decide on different values $v_1$ and $v_2$, respectively. We show by contradiction that this is not possible.

Uniform Validity states that for $p_i$ to have decided on $v_1$, at least one correct process must have proposed $v_1$. Similarly, for $p_j$ to have decided on $v_2$, at least one correct process must have proposed $v_2$. However, all correct processes will propose the same value, because of the assumption of deterministic processing. Hence $v_1 = v_2$, a contradiction. $\square$

**Theorem 3.7:** Every **correct** process proposes a value at most once. *(Propositional Integrity)*

**Proof (Sketch):** A correct process $p_i$ proposes a value by calling the *propose-now()* function. The function updates $V_i[i]$ to a non-null value returned by the *giv()* function. Once $V_i[i]$ is set to a non-null value, it never becomes null again. Since the *propose-now()* function is invoked in Block 5 and Block 6 only if $V_i[i]$ is null, it follows that $p_i$ proposes a value at most once. $\square$

**Theorem 3.8:** For any two elements $\mathcal{P}$ and $\mathcal{Q}$ of $\mathcal{PC}$ and for two correct processes $p$ and $q$ such that $p \in \mathcal{P} \wedge p \notin \mathcal{Q}$ and $q \in \mathcal{Q} \wedge q \notin \mathcal{P}$, if both $p$ and $q$ propose a value, then at least one of the following is true:

- at least one process in $\mathcal{P}$ is suspected by some other process in $\mathcal{P}$.
- at least one process in $\mathcal{Q}$ is suspected by some other process in $\mathcal{Q}$. *(Weak Byzantine Laziness)*

**Proof (Sketch):** Let us assume that neither (I) nor (II) is true. We prove that this is not possible by contradiction.

In the context of the LBC-Psync algorithm, when a correct $pc$ member "suspects" a fellow $pc$ member, it initiates the next $pc$ reselection round. Hence, the situations in which a $pc$ member suspects a fellow $pc$ member are the three cases given in Observation 1. Conversely, a correct $pc$ member initiates the next $pc$ reselection round only if it suspects a fellow $pc$ member due to one of those three cases.

We can assume, without loss of generality, that $\mathcal{P}$ precedes $\mathcal{Q}$ in the reselection rounds. That is, if $committee(n_1) = \mathcal{P}$ and $committee(n_2) = \mathcal{Q}$, then $n_1 < n_2$.

A correct process proposes a value by invoking the *propose-now()* function in Block 5 and Block 6 only if it is currently a $pc$ member. That is, $p$ proposes its value only when $\mathcal{P}$ is the $pc$ (since $p \notin \mathcal{Q}$), and $q$ proposes its value only when $\mathcal{Q}$ is the $pc$ (since $q \notin \mathcal{P}$). Since $q$ proposes a value, it must be a $pc$ member, which means that reselection round number $n_2$ must have been initiated by some member of the $pc$ corresponding to reselection round number $n_2 - 1$. By induction on the difference $n_2 - n_1$, we can see that for the reselection round number $n_2$ to be initiated, the reselection round number $n_1 + 1$ must have been initiated some time before. However, only an element $r$ of $\mathcal{P}$ can initiate the reselection round number $n_1 + 1$. Depending on whether $r$ is correct, we will have one or the other of the following two situations:

(i) If $r$ is correct, then it must have initiated the reselection round number $n_1 + 1$ only because it suspects some other member of $\mathcal{P}$.
(ii) If $r$ is faulty and was the first member of $\mathcal{P}$ to send a *reselect* message for the reselection round number $n_1 + 1$. Then, any correct process that receives this message will either send its own *reselect* message or forward $r$'s *reselect* message to all the processes. Thus, $p$ will come to suspect $r$ from case (3) in Observation 1.

In both (i) and (ii) above, condition (I) is true, which is a contradiction. □

## 3.7 Safety and Liveness of the Semi-Passive Replication Algorithm

In this section, we show that if our semi-passive replication algorithm instantiates a sequence of LBC-Psync algorithms (one for each client request, with the next instance created after the current instance terminates), then the semi-passive replication algorithm satisfies the specifications given in Section 2.2.

Termination of the replication algorithm follows directly from the termination of the LBC-Psync algorithm. From Theorem 3.3, the client will eventually receive responses from each of the correct replicas. From Theorem 3.6, it can be seen that all correct replicas send identical responses. Since there are at least $t + 1$ correct replicas, the client will eventually receive at least $t + 1$ responses with identical result and accept that result.

The Total Order property of the replication algorithm follows from 1) the fact that the $k^{th}$ client request at all the replicas is the same, 2) and the Agreement property of the LBC-Psync algorithm.

Update Integrity of the replication algorithm can be shown as follows: A correct replica $p_i$ will execute $upd_k^i$ only after a decision has been reached in the $k^{th}$ instance of the LBC-Psync algorithm. However, for a decision to be reached, at least $t+1$ processes must have proposed an initial value (they must have processed the request $req_k$). At least one of those $t + 1$ processes must be correct. A correct process will process a request only if it receives the request. Hence, it is clear that $p_i$ executes $upd_k^i$ only if the client sent $req_k$. After executing $upd_k^i$, replica $p_i$ will remove $req_k$ from its *receive queue* and add the request to the *handled set* (see Section 2.4). Thus, the request will not be handled again, and hence $upd_k^i$ will be executed at most once by $p_i$.

Response Integrity of the replication algorithm can be shown as follows: For any response to be accepted by the client, at least one correct replica must have processed the request, proposed an initial value for consensus, and decided on that value. After deciding on the value, the correct replica will update its internal state. Therefore, for any response $resp_k$ corresponding to the request $req_k$ *accepted* by the client, $upd_k^i$ will be executed by some correct replica $p_i$.

The Weak Byzantine Parsimony property of the replication algorithm follows from the Weak Byzantine Laziness property of the LBC-Psync algorithm, since "processing a request" at the replication algorithm level is equivalent to "proposing a value" at the LBC-Psync algorithm level.

## 3.8 Factors Influencing the Efficiency of the LBC-Psync Algorithm

As proved in Section 3.6, if at some point after the GST, the $pc$ consists only of correct members, then a decision will be reached. The maximum number of reselection rounds after GST (let us call this $max\text{-}rounds$) required for an element of $\mathcal{PC}$, say $P$, that consists entirely of correct processes to become the primary committee is an important measure of the efficiency of the LBC algorithm. $max\text{-}rounds$ gives the upper bound on the number of reselection rounds after GST before a correct process can make a decision. In this section, we present examples to illustrate the factors that determine $max\text{-}rounds$.

Consider the case in which the fault resilience is optimal, i.e., $n = 2t + 1$. Consider a reselection policy such that the set $\mathcal{PC} = \mathcal{S}^{t+1}$, $|\mathcal{PC}| = C^{2t+1}_{t+1}$, and each element of $\mathcal{PC}$ is chosen exactly once in $C^{2t+1}_{t+1}$ reselection rounds in some order that is identical across all replicas. It can be seen that this reselection policy satisfies Assumption 1. In this case, although the fault resilience is optimal, $max\text{-}rounds$ is rather large and is equal to $C^{2t+1}_{t+1}$. This value for $max\text{-}rounds$ is obtained by considering the worst case of $t$ faults, where there is only one element of $\mathcal{PC}$ that consists entirely of correct processes, and that element is chosen in the last of $C^{2t+1}_{t+1}$ reselection rounds.

We now obtain the value of $max\text{-}rounds$ for a more general case. Let $n$ be any value greater than $2t$. Let $f$ be the actual number of faulty processes. Consider a reselection policy such that the set $\mathcal{PC} = \mathcal{S}^{t+1}$, and each element of $\mathcal{PC}$ is chosen exactly once in $|\mathcal{PC}|$ reselection rounds in some order that is identical across all replicas. Then, $max\text{-}rounds = [C^{n}_{t+1} - C^{n-f}_{t+1}]$. Here, $C^{n-f}_{t+1}$ is the number of elements of $\mathcal{PC}$ that consist entirely of correct processes. In the worst case, the reselection policy chooses all the elements of $\mathcal{PC}$ that have at least one faulty process, before choosing any of the $C^{n-f}_{t+1}$ elements that consist entirely of correct processes.

Now, consider another case in which the reselection policy is implemented by having $committee(x)$ return the set $\{p_{x \ mod \ n}, p_{(x+1) \ mod \ n}, \cdots, p_{(x+t) \ mod \ n}\}$ as the $pc$ for the reselection round $x$. In the worst case, the $t$ corrupted processes could be at "distance" $t$ from each other ($\{p_t, p_{2t+1}, \cdots, p_{t^2+t-1}\}$). The minimum value of $n$ required for the reselection policy to satisfy Assumption 1 is $n = t^2 + t + 1$. If $f$ is the number of actual faults ($f \leq t$), then $max\text{-}rounds = f \cdot t + f$. Specifically, if $f = t$, then $max\text{-}rounds = t^2 + t$.

We have presented a few examples that show that the reselection policy, the number of processes, the maximum number of faulty processes, and the actual number of faulty processes are all factors that influence the efficiency of the LBC-Psync algorithm. By careful selection of the reselection policy, it is possible to tune the algorithm for desired levels of resilience or efficiency.

## 3.9 Latency Degree

In [12], Schiper introduced the concept of *latency degree* to measure the efficiency of a distributed algorithm. Informally, the latency degree of a consensus algorithm is the minimum number of communication steps over all possible runs, which is typically obtained in *good runs*. A good run is a run with no faults and

no suspects. Our LBC-Psync algorithm has a latency degree of 2. There are only two communication steps in a good run (as shown in Figure 2(a)):
1. A $pc$ member sends its *propose* message with its initial value to all.
2. Once a process has received *propose* messages with identical initial values from all the $(t+1)$ $pc$ members, it sends a *decision* message to all, and decides.

A latency degree of 2 is optimal [8]. Hence, our LBC-Psync algorithm is optimal in the number of communication steps in good runs.

## 3.10 Optimizations

We now present some optimizations that can be used to improve the performance of the LBC-Psync algorithm (and hence the semi-passive replication algorithm).

From the point of view of the semi-passive replication algorithm, the overhead involved to "settle" on an all-correct-$(t+1)$-subset (as the $pc$) after GST could easily be made a one-time overhead, by carrying over the knowledge of the latest primary committee from one instance of the consensus problem to the next instance of the consensus problem.

A second optimization can be used to restrict the number of rounds taken to reach decision to $O(t)$ for any $n > 2t$. First, we observe that a process can make a decision as soon as it gets $(t+1)$ *propose* messages with identical values. Second, we observe that the *propose* messages sent by any correct process will have the same value as the *propose* message of any other correct process. These observations mean that a decision could be reached as soon as $(t+1)$ correct processes send their *propose* messages. For that reason, instead of needing to wait to "settle" on an all-correct-$(t+1)$-subset, it is enough to perform the minimum number of $pc$ reselections required to "cover" $(t+1)$ correct processes. Suppose we have a reselection policy such that the set $\mathcal{PC} = \mathcal{S}^{t+1}$, and each element of $\mathcal{PC}$ is chosen exactly once in $|\mathcal{PC}|$ reselection rounds in some order that is identical across all replicas. It can be easily seen that irrespective of the value of $n$ (as long as $n > 2t$), the minimum number of $pc$ reselections required to "cover" $(t+1)$ correct processes is only $2t+1$. Hence, if we require that a process $p_i$ upon becoming a $pc$ member for the first time, immediately obtain its initial value and send a *propose* message, then a decision can be reached at the end of $(2t+1)$ $pc$ reselection rounds. However, if the algorithm is made to terminate as soon as a decision has been reached, then this optimization should not be used in conjunction with the first optimization, because at termination, it may be that not all *reselect* messages have been delivered. Hence the $pc$ information should not be carried over to the next instance of the consensus problem; rather, each instance of the consensus problem should start from the same $pc$.

## 4 Lazy Byzantine Consensus in an Asynchronous System with Unreliable Fault Detectors for Byzantine Faults

In this section, we present a algorithm that solves the lazy Byzantine consensus problem in the asynchronous system model augmented with unreliable fault detectors for Byzantine faults [10].

### 4.1 System Model

The system consisting of the $n$ server processes, $\mathcal{S}$, is an asynchronous system with no bounds on message delivery times and relative processor speeds. The server processes communicate with each other through reliable communication channels. *Reliable* means that messages exchanged between correct processes are eventually received and are not modified by the underlying communication medium. We do not consider network partitions. Processes have access to local clocks, which are not synchronized.

Processes could be correct or faulty. A correct process conforms to its specification, and a faulty process can behave arbitrarily. We allow up to $t$ processes to be faulty, and require that $t < \frac{n}{3}$. Thus, at least $\lceil \frac{2n+1}{3} \rceil$ processes are correct. We assume authenticated communication so that messages can be signed with the name of the sending process in such a way that the signature cannot be forged by any other process. Thus, our fault model is the Authenticated Byzantine fault model [6].

### 4.2 Unreliable Fault Detectors for Byzantine Faults

In [10], Kihlstrom et al. defined the $\Diamond S$(Byz) class of fault detectors for Byzantine faults. Here, we briefly describe their work and propose an extension to define a new $\Diamond S$(LazyByz) class of fault detectors that can be used for solving lazy Byzantine consensus.

Kihlstrom et al. identified the following completeness and accuracy properties of fault detectors for solving consensus in a Byzantine environment:

**Eventual Strong Byzantine Completeness** There is a time after which every process in $\mathcal{S}$ that has exhibited a detectable Byzantine fault is permanently suspected by every correct process.

**Eventual Weak Byzantine $(t+1)$-Completeness** There is a time after which every process in $\mathcal{S}$ that has exhibited a detectable Byzantine fault is permanently suspected by at least $t+1$ correct processes.

**Eventual Weak Accuracy** There is a time after which some correct process in $\mathcal{S}$ is never suspected by any correct process.

The $\Diamond W$(Byz) class of Eventually Weak Byzantine fault detectors satisfies the Eventual Weak Byzantine $(t+1)$-Completeness and the Eventual Weak Accuracy properties above. The $\Diamond S$(Byz) class of Eventually Strong Byzantine fault detectors satisfies the Eventual Strong Byzantine Completeness and the Eventual Weak Accuracy properties.

In the properties defined above, the term *detectable* Byzantine fault is used to indicate omission and commission faults. An omission fault occurs when a process does not send a required message to one or more correct processes. A commission fault occurs either when 1) a process sends messages that contain a valid signature but are improperly formed or contain improper proofs, or 2) a process sends two or more *mutant* messages. Two or more messages are said to be mutants if they are of the proper form, have proper proofs, have the same source, and have the same round/phase, but have different contents. If a correct process $p_i$ has noticed that another process $p_j$ is exhibiting a commission fault, then $p_i$ will permanently suspect $p_j$. Process $p_i$ will also send to all other processes the proof that $p_j$ has exhibited a commission fault; then, all correct processes will receive this proof, declare $p_j$ to be Byzantine-faulty, and permanently suspect $p_j$. Detectable Byzantine faults are different from non-detectable Byzantine faults in that non-detectable Byzantine faults cannot be attributed to a particular process and are not observable by a process based on the messages it receives.

In [10], Kihlstrom et al. showed 1) that $\diamondsuit W$(Byz) fault detectors are the weakest class of failure detectors that can be used to solve Byzantine consensus in asynchronous systems, and 2) that a $\diamondsuit W$(Byz) fault detector can be transformed into a $\diamondsuit S$(Byz) fault detector. They presented a three-phased consensus algorithm based on the rotating coordinator (primary process) paradigm, in which termination is guaranteed by the Eventual Weak Accuracy property. Intuitively, we can see that consensus will be reached when the correct process that is not suspected by any other correct process becomes the coordinator.

However, in lazy Byzantine consensus, we do not have a single coordinator, but a $(t+1)$ committee of primary processes. The Eventual Weak Accuracy condition stated above does not ensure termination even if the reselection policy guarantees that an element of $\mathcal{PC}$ (say $P$) consisting entirely of correct processes will become the $pc$ infinitely often (as stated by Assumption 1). The reason is that the Eventual Weak Accuracy property only guarantees that (eventually) some correct process will never be suspected by any correct process. The other $t$ correct members of $\mathcal{P}$ could be suspected forever by enough other members such that the lazy Byzantine consensus algorithm might never terminate. Hence, we define a new Eventual Weak Lazy Byzantine Accuracy property as follows:

**Eventual Weak Lazy Byzantine $(t+1)$-Accuracy** There is a time after which none of the processes in some set $\mathcal{P} \in \mathcal{PC}$ are ever suspected by a majority of correct processes.

We define $\diamondsuit S$(LazyByz) to be the class of unreliable fault detectors that satisfy Eventual Strong Byzantine Completeness and Eventual Weak Lazy Byzantine $(t+1)$-Accuracy.

## 4.3 A Lazy Byzantine Consensus Algorithm using $\diamondsuit S$(LazyByz)

We call this algorithm the *LBC-$\diamondsuit S$(LazyByz) algorithm*. The LBC-$\diamondsuit S$(LazyByz) algorithm is very similar to the LBC-Psync algorithm, except for the following differences:

1. The LBC-$\diamondsuit S$(Byz) algorithm does not directly employ timeouts, as the timeouts are abstracted away

in the unreliable fault detector. Hence, the algorithm makes no calls to the *scheduleTimeout()* and *cancelTimeout()* functions.

2. In the LBC-psync algorithm, one of the three triggers that will cause a $pc$ member $p_i$ to initiate the next $pc$ reselection round is the occurrence of a timeout before $p_i$ has received the *propose* message from all the other $pc$ members with the same initial value as its own (case 1 in Observation 1). In the LBC-$\diamondsuit S$(LazyByz) algorithm, we use the following trigger instead: if $pc$ member $p_i$ finds that any other $pc$ member is currently suspected by $\lfloor\frac{n+t}{2}\rfloor$+1 distinct processes possibly including $p_i$ (which is equivalent to a majority of correct processes), then it initiates the next $pc$ reselection round.

3. An additional trigger for $p_i$ to initiate the next $pc$ reselection round is the exhibition of a commission fault by a fellow $pc$ member. If that happens, $p_i$ does not have to wait for suspects from $\lfloor\frac{n+t}{2}\rfloor$+1 processes.

Hence, observation 1 in the LBC-Psync algorithm could be restated for the LBC-$\diamondsuit S$(LazyByz) algorithm as follows:

**Observation 2** *There are only four cases in which a correct $pc$ member $p_i$ will initiate the next $pc$ reselection round. They are (1) a fellow $pc$ member is currently suspected by $\lfloor\frac{n+t}{2}\rfloor$+1 distinct processes, (2) the initial value of some fellow $pc$ member differs $p_i$'s own initial value, (3) a fellow $pc$ member has exhibited a commission fault, or (4) $p_i$ finds that a fellow $pc$ member has already reselected a new $pc$.*

### 4.4 Implementing a $\diamondsuit S$(LazyByz) Fault Detector

In [10], Kihlstrom et al. described the implementation of a $\diamondsuit S$(Byz) class Byzantine fault detector under partial synchrony assumptions [6] at each process $p_i$ that outputs a list of processes that $p_i$ currently suspects of having exhibited a detectable Byzantine fault. We assume a similar implementation of a $\diamondsuit S$(LazyByz) fault detector module in each of our server replicas. We also require another easily implementable feature, namely that when the fault detector module at a process $p_i$ begins to suspect a process $p_j$, process $p_i$ will send a signed *added-suspect($p_j$)* message to all processes. Later, when the fault detector module at $p_i$ stops suspecting process $p_j$ (maybe temporarily), process $p_i$ will send a *removed-suspect($p_j$)* message to all processes. If $p_i$ has detected that $p_j$ exhibited a commission fault, it will send an *added-suspect($p_j$)* message, but it will never send a *removed-suspect($p_j$)* message. $p_i$ will also send the proof that $p_j$ has exhibited a commission fault to all the processes.

We also follow Kihlstrom's method [9] for masking a fault in which a Byzantine process sends a message to some correct process but not to others. We do so by requiring that any correct process that receives a message for the first time must immediately relay it to all processes.

Additionally, we require that each process maintain a bit matrix of suspects called the *suspect-matrix*. The cell *suspect-matrix[i][j]* of the matrix at a process $p_k$ is set when $p_k$ receives an *added-suspect($p_j$)*

message from process $p_i$; the cell is reset when $p_k$ receives a *removed-suspect($p_j$)* message from process $p_i$. The *added-suspect* and *removed-suspect* messages will be signed by the sender and timestamped with the sender's local clock. That means that if an *added-suspect($p_j$)* from process $p_i$ is forwarded to process $p_k$ (by some process other than $p_i$), but a more recent (determined from the timestamp) *removed-suspect($p_j$)* has been received at $p_k$, then $p_k$ will ignore the *added-suspect($p_j$)* message. Thus, process $p_k$ can determine the number of processes currently suspecting $p_j$ (based on the messages received so far) by checking the $j^{th}$ column of the matrix. If $p_k$ and $p_j$ are current $pc$ members, and $p_k$, based on its *suspect-matrix* finds that $p_j$ is suspected by $\lfloor \frac{n+t}{2} \rfloor$+1 distinct processes, then $p_k$ will initiate the next $pc$ reselection round.

In the LBC-Psync algorithm, it was the responsibility of the correct member(s) in the $pc$ to ensure progress by initiating the next $pc$ reselection round, if the $pc$ member has not received the initial value from some faulty $pc$ member. The correct $pc$ member relies on local timeouts to decide whether it has waited "long enough" before it initiates the next $pc$ reselection round. In the LBC-$\Diamond S$(LazyByz) algorithm, again, only a current $pc$ member $p_i$ can initiate the next $pc$ reselection round. However, to detect omission faults of a fellow $pc$ member, $p_i$ relies not only on its local fault detector module, but also on the messages from the fault detector modules at other processes. For that purpose, when any process $p_i$ ($pc$ member or not) switches to a new $pc$, it schedules a timeout by which it expects to get a *propose* message from all $pc$ members that have not hitherto sent such a message. If $p_i$ is a $pc$ member and has not sent its *propose* message before, it sends the message before scheduling the timeout. If any $pc$ member $p_j$ does not send its *propose* message before the timeout expires, then that member will be suspected by the fault detector module, and a signed *added-suspect($p_j$)* message will be sent to all the processes. If the *propose* message sent by a correct $p_j$ is delayed in reaching $p_i$, either because $p_j$ is temporarily slow or because the message transmission is slow, then $p_i$ may incorrectly come to suspect $p_j$. Such incorrect suspicions are allowed by the properties of the fault detector. If $p_j$ acts in a timely manner in later $pc$ reselection rounds, $p_i$ will remove the suspicion for $p_j$. As in the LBC-Psync algorithm, a timeout set for the current $pc$ is cancelled when a decision is reached or when a new $pc$ reselection round is initiated.

## 4.5 Safety and Liveness of the LBC-$\Diamond S$(LazyByz) Algorithm

The proofs to show that the LBC-$\Diamond S$(LazyByz) algorithm satisfies the Uniform Integrity, Uniform Validity, Agreement, and Propositional Integrity properties are identical to the corresponding proofs for the LBC-Psync algorithm.

The termination of the LBC-Psync algorithm is conditional upon the existence of GST. Similarly, the termination of the LBC-$\Diamond S$(LazyByz) algorithm is conditional upon the Eventual Weak Lazy Byzantine ($t+1$)-Accuracy property of the $\Diamond S$(LazyByz) fault detector. Proceeding as we did in Lemma 3.1, we can show using Observation 2 that if there is a time $\tau$ after which all the processes in some set $\mathcal{P} \in \mathcal{PC}$ are never suspected by a majority of correct processes, then no further $pc$ reselection rounds will occur. By Assumption 1 of the reselection policy, $\mathcal{P}$ must become the primary committee some time after $\tau$. This implies that the number of $pc$ reselection rounds after $\tau$ will be finite. Thus, each process will eventually decide on some value.

The Weak Byzantine Laziness property of the LBC-$\Diamond S$(LazyByz) algorithm can be proved much like the LBC-Psync algorithm, but using Observation 2 instead of Observation 1.

### 4.6 Efficiency, Latency Degree, and Optimizations

In Section 3.8, we observed that the reselection mechanism, the maximum number of faulty processes, the actual number of faulty processes, and the number of processes are the factors that influence the efficiency of the LBC-Psync algorithm. The same observation applies to the LBC-$\Diamond S$(LazyByz) algorithm (with the exception that $n$ must be greater than $3t$ for the LBC-$\Diamond S$(LazyByz) algorithm). Like the LBC-Psync algorithm, the LBC-$\Diamond S$(LazyByz) algorithm has a latency degree of 2. The optimizations presented in Section 3.10 can also be applied to the LBC-$\Diamond S$(LazyByz) algorithm.

## 5 Non-Determinism

One of the advantages of passive replication over active replication in the crash fault model is that passive replication does not require the replicas to be deterministic. However, in our Byzantine fault-tolerant semi-passive replication algorithm, because we use a primary committee (instead of a single primary process as in the crash fault model), the replicas need to be deterministic. In particular, the initial values of all correct replicas for a particular instance of the lazy Byzantine consensus algorithm need to be the same. That is a drawback, since there are many services that are non-deterministic. For example, if the replicas want to reach a decision on the time at which a particular transaction took place (based on a client query), their initial values could differ if each replica obtains the time only from its local clock.

To tackle such situations, we follow an approach similar to that of [2], and add an extra communication step to our consensus protocol: the $pc$ members first send their initial values (that could be different across correct $pc$ members) to all the processes. Then, each $pc$ member chooses a *consolidated* initial value by applying a deterministic function on the set of values obtained from all the $pc$ members (e.g., the average of values). This *consolidated* initial value will be the value that is sent in the *propose* message. In order to prevent the situation in which the $pc$ consists of $t$ faulty members that faithfully send their messages in time, but try to bias the *consolidated* initial value, a correct $pc$ member could initiate the next $pc$ reselection round if it finds that the *consolidated* initial value is not within an $\epsilon$ bound of its original initial value. The value of $\epsilon$ should be given by the service specification.

## 6 Conclusions

In this paper, we presented specifications for Byzantine fault-tolerant semi-passive replication and lazy Byzantine consensus. We described an algorithm for Byzantine fault-tolerant semi-passive replication based on a series of lazy Byzantine consensus algorithms. We then presented the LBC-Psync algorithm for lazy Byzantine consensus in the partial synchrony model of [6] and the LBC-$\diamondsuit S$(LazyByz) algorithm for lazy Byzantine consensus in an asynchronous system augmented with unreliable fault detectors. We specified an extension to Kihlstrom et al.'s $\diamondsuit S$(Byz) class of unreliable fault detectors to solve lazy Byzantine consensus.

We also proved that our consensus algorithms provide safety and liveness. Our algorithms are optimal in good runs, having a latency degree of 2. We also used examples to show, how the reselection policy can be appropriately chosen so as to tune the algorithm for either optimal fault resilience or efficiency in the presence of faults. We presented optimizations to improve the performance of the algorithms.

## References

[1] N. Budhiraja, F. Schneider, S. Toueg, and K. Marzullo. *The Primary-Backup Approach*, pages 199–216. ACM Press - Addison Wesley, 1993.

[2] M. Castro and B. Liskov. Practical Byzantine Fault Tolerance. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, pages 173–186, February 1999.

[3] X. Défago and A. Schiper. Specification of Replication Techniques, Semi-Passive replication, and Lazy Consensus. Technical Report IC-2002-07, EPFL, Switzerland, February 2002.

[4] X. Défago, A. Schiper, and N. Sergent. Semi-Passive Replication. In *Proceedings of the 17th Symposium on Reliable Distributed Systems (SRDS-17)*, pages 43–50, October 1998.

[5] Xavier Défago. *Agreement-Related Problems: From Semi-Passive Replication to Totally Ordered Broadcast*. PhD thesis, EPFL, Switzerland, August 2000.

[6] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the Presence of Partial Synchrony. *Journal of the ACM (JACM)*, 35(2):288–323, April 1988.

[7] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM*, 32(2):372–382, April 1985.

[8] R. Guerraoui and A. Schiper. Consensus: The Big Misunderstanding. In *Proc. 6th IEEE Computer Society Workshop on Future Trends in Distributed Computing Systems (FTDCS-6)*, pages 183–188, 1997.

[9] K. P. Kihlstrom. *Survivable Distributed Systems: Design and Implementation*. PhD thesis, University of California, Santa Barbara, 1999.

[10] K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith. Solving Consensus in a Byzantine Environment Using an Unreliable Fault Detector. In *Proceedings of the International Conference on Principles of Distributed Systems (OPODIS)*, pages 61–75, December 1997.

[11] D. Malkhi and M. Reiter. Unreliable Intrusion Detection in Distributed Computations. In *Proceedings of the 10th Computer Security Foundations Workshop (CSFW97)*, pages 116–124, Rockport, MA, June 1997.

[12] A. Schiper. Early Consensus in an Asynchronous System with a Weak Failure Detector. *Distributed Computing*, 10(3):149–157, 1997.

[13] F. B. Schneider. *Replication Management Using the State Machine Approach*, pages 169–197. ACM Press - Addison Wesley, 1993.