

BEE'S STRATEGY AGAINST BYZANTINES  
Replacing Byzantine Participants

by

**Roberto Baldoni, Silvia Banomi, Shlomi Dolev, Michel Raynal, Amitay Shaer**

Technical Report #18-05

September 21, 2018

The Lynne and William Frankel Center for Computer Science  
Department of Computer Science,  
Ben-Gurion University, Beer Sheva, Israel

# BEE'S STRATEGY AGAINST BYZANTINES

## Replacing Byzantine Participants <sup>★</sup>

(Preliminary Version)

Amitay Shaer<sup>1</sup>, Shlomi Dolev<sup>1</sup>, Silvia Bonomi<sup>2,3</sup>, Michel Raynal<sup>4</sup>, and Roberto Baldoni<sup>2,3</sup>

<sup>1</sup> Dept. of Computer Science, Ben-Gurion University of the Negev, Beer-Sheva, Israel  
`{shaera, dolev}@cs.bgu.ac.il`

<sup>2</sup> Research Center of Cyber Intelligence and Information Security (CIS)

<sup>3</sup> Department of Computer, Control, and Management Engineering "A. Ruberti",  
Sapienza Università di Roma, Via Ariosto 25, 00185 Roma, Italy  
`{bonomi,baldoni}@diag.uniroma1.it`

<sup>4</sup> IRISA, Université de Rennes, 35042 Rennes, France  
`michel.raynal@irisa.fr`

**Abstract.** Schemes for the identification and replacement of two-faced Byzantine processes are presented. The detection is based on the comparison of the (blackbox) decision result of a Byzantine consensus on input consisting of the inputs of each of the processes, in a system containing  $n$  processes  $p_1, \dots, p_n$ . Process  $p_i$  that received a gossiped message from  $p_j$  with the input of another process  $p_k$ , that differs from  $p_k$ 's input value as received from  $p_k$  by  $p_i$ , reports on  $p_k$  and  $p_j$  being two-faced. If enough processes (where enough means at least  $t + 1$ ,  $t < n$  is a threshold on the number of Byzantine participants) report on the same participant  $p_j$  to be two-faced, participant  $p_j$  is replaced. If less than the required  $t + 1$  processes threshold report on a participant  $p_j$ , both the reporting processes and the reported process are replaced. If one of them is not Byzantine, its replacement is the price to pay to cope with the uncertainty created by Byzantine processes. The scheme ensures that any two-faced Byzantine participant that prevents fast termination is eliminated and replaced. Such replacement may serve as a preparation for the next invocations of Byzantine agreement possibly used to implement a replicated state machine.

**Keywords:** Distributed Algorithms · Consensus · Byzantine Failures · Detection.

---

<sup>★</sup> The research was partially supported by the Rita Altura Trust Chair in Computer Sciences; the Lynne and William Frankel Center for Computer Science; the Ministry of Foreign Affairs, Italy; the grant from the Ministry of Science, Technology and Space, Israel, and the National Science Council (NSC) of Taiwan; the Ministry of Science, Technology and Space, Infrastructure Research in the Field of Advanced Computing and Cyber Security; Michel Raynal was visiting BGU with the support of Dozor founding; and the Israel National Cyber Bureau. Contact author: Shlomi Dolev.

## 1 Introduction

The Byzantine Agreement (BA) problem, introduced by Pease, Shostak, and Lamport in [7], is known as a fundamental problem in fault-tolerant distributed computing. The problem has received a lot of attention in the literature and has become the essence of a variety of schemes in distributed computing.

Solving the Byzantine agreement problem is not necessarily tied to the detection of the Byzantine processes, the only success criterion of the agreement is whether all correct processes agree on the same value. In this work, we focus on detecting a sub-class of Byzantine behavior (i.e., namely *two-faced* Byzantine processes) for the sake of replacing them in the next Byzantine agreement invocation. The detection and replacement procedure can be plugged in any other algorithm using a Byzantine Agreement e.g., possibly as part of the implementation of a replicated state machine. The final goal is to prepare for the future invocations of the agreement, trying to ensure that the next invocation of the Byzantine agreement will cope with a smaller number of two-faced Byzantine processes.

**The Byzantine Agreement Problem.** In the Byzantine Agreement Problem, there are  $n$  processes,  $\Pi = \{p_1, \dots, p_n\}$  with unique names over  $N = \{1, \dots, n\}$  and at most  $t < n$  of the processes can be Byzantine. Each process starts with an input value  $v$  from a set of values  $V^1$ . The goal is to ensure that all non-faulty processes eventually output the same value. The output of a non-faulty process is called the *decision value*.

More formally, an algorithm solves the Byzantine Agreement if the following conditions hold:

- **Agreement.** All non-faulty processes agreed on the same value (i.e., there are no two non-faulty processes that decide different values).
- **Validity.**<sup>2</sup> If all non-faulty processes start with the same value  $v$ , the decision value of all non-faulty participants is  $v$ .
- **Termination.** Eventually, all non-faulty processes decide a value.

Reaching agreement in presence of Byzantine processes is expensive as the number of messages grows quadratically with the number of participants  $n$  and the number of rounds (time) grows linearly with the number of Byzantine participants  $t$  (with  $n > 3t$ ).

Applications may repeatedly invoke agreement instances (e.g., as part of implementing a replicated state machine). Typically, the presence of Byzantine activity is rare, and it is desirable that the overhead in handling Byzantine activity will be tuned to the actual situation. Hence, we want to adjust the time it takes for each consensus invocation to run according to the actual situation at the time. In addition, when the system is interactive and never stops (as in the

<sup>1</sup> Binary Agreement is defined with the set  $V = \{0, 1\}$ .

<sup>2</sup> There is an alternative, stronger property for *Validity* [6]. The decision value  $v$  has to be an input value of at least one non-faulty process.

case of a replicated state machine) the number of Byzantine participants may be accumulated to exceed any threshold. To avoid the accumulation of Byzantine participants over time, we suggest a detection and replacements of Byzantine participants. Thus, we suggest to couple the consensus algorithm with a detection mechanism. As soon as a Byzantine activity is detected, the suspected Byzantine process will be eliminated and replaced.

The rest of the paper is organized as follows: in Section 2, we describe shortly our system settings. In Section 3, we introduce our approach of the detection process as a combination of two algorithms, *fast* and *slow* presented in Section 4 and 5 respectively.

## 2 System Settings

We consider a distributed system composed of  $n$  processes, each having a unique identifier  $p_1, p_2, \dots, p_n$ . We assume that up to  $t$  processes can be Byzantine with  $n > 3t$ . A process is said to be *Byzantine* if it deviates from the protocol, otherwise it is said to be *correct*. Processes communicate through message exchanges. A reliable communication is assumed, where a point-to-point channel exists for every pair of processes. More precisely, if a correct process  $p_i$  sends a message  $m$  to a process  $p_j$ , then  $m$  will be delivered by process  $p_j$ . Channels are authenticated, i.e., when a process  $p_j$  receives a message  $m$  from a process  $p_i$ ,  $p_j$  knows that  $m$  has been generated by  $p_i$ .

We consider a particular sub-class of Byzantine failures called *two-faced Byzantine*. A process is said to be *two-faced Byzantine* if it is supposed to send a broadcast message to all processes in the system but it is sending different messages to different processes. In the sequel, we use the term *process acting as Byzantine* to refer to a process that exhibits two-faced behaviour.

The system is synchronous and evolves in sequential synchronous rounds  $r_1, r_2, \dots, r_i, \dots$ . Every round is divided into three steps: (i) *send step* where the processes send all the messages for the current round, (ii) *receive step* where the processes receive all the messages sent at the beginning of the current round<sup>3</sup> and (iii) *computation step* where the processes process the received messages and prepare new messages to be sent in the next round. The processes have the ability to make speculative execution when needed, meaning an execution is made, but all states are saved until a condition holds, sometimes forcing the system rolling back to the previous state (see e.g., [5]).

We assume the existence of a trusted entity component (or components), called *hypervisor*. The *hypervisor* can only receive messages from the processes but cannot send them messages. The *hypervisor* can eliminate or replace a process with a different process instance. The *hypervisor* is assumed to be correct all the time. We consider two different types in which the hypervisor integrated into the system:

<sup>3</sup> Let us note, that in round-based computations, all `deliver()` events happen during the receive step.

- **Global hypervisor.** There exists just one global hypervisor that controls all the processes in the system.
- **Local hypervisor.** There is a hypervisor that controls  $p$  for each process  $p$ . Each local hypervisor can communicate with the other local hypervisors.

### 3 Byzantine Detection and Replacement

Many various algorithms exist for solving different problems given Byzantine processes (some of the algorithms require a restriction on the number of Byzantine participants). In order to detect Byzantine processes, it is necessary that participants running the distributed algorithm report and audit activities performed by each other. Let us observe that given  $t + 1$  or more testimonies for process  $p_j$  as faulty,  $p_j$  is Byzantine as there exists at least one correct process suspecting  $p_j$  with correct evidences (i.e., there exists at least one correct process that observed a misbehavior from  $p_j$ ). On the other hand, in case there are at least one and less than  $t + 1$  testimonies on  $p_j$  being Byzantine, then at least one process is Byzantine among the reporting processes and  $p_j$ . Unfortunately, in this case it is not possible to detect who is the Byzantine process but it is possible to identify a group of processes that collectively exhibits a bad behavior and that for sure contains the Byzantine process.

In the heart of the detection, we expect two main parts: fast and slow. The fast part is run by the processes as long as there is no Byzantine activity. When Byzantine activity is discovered the slow part takes place and replaces the Byzantine processes that acted in a two faced fashion, enforcing the system to continue beyond the fast part. We use the term *fast termination* for a scenario in which the fast part is successfully completed.

Thus, our problem can be summarized by adding the following properties to the specification of the original Byzantine agreement problem:

- **Completeness.** A process  $p_i$  acting two-faced in a manner that eliminates fast termination is suspected by some correct process  $p_j$ .
- **Sacrifice.** If process  $p_i$  is suspected by at least  $t + 1$  processes,  $p_i$  is the only one to be replaced. Otherwise,  $p_i$  and the processes that suspect  $p_i$  are replaced.

Note, that the *completeness* above somewhat resembles game theory consideration, enforcing Byzantine process that would like to survive to allow the system to terminate fast, and the best global utility is achieved.

### 4 Byzantine Free Fast Termination

We suggest a Byzantine agreement protocol composed of *fast* and *slow* parts. As long as there is no indication of a Byzantine activity, such that is causing the slow algorithm to be executed, only the fast algorithm takes place (Algorithm 1). Combined with the capability of speculative execution and roll back, the actual

execution is only two rounds (rounds 1 and 2 of Phase 1). Contrarily, when a testimony of Byzantine activity has been discovered, processes start to execute the slow algorithm (Algorithms 3 or 4).

Optimistically, all processes are assumed to be correct and start with the fast algorithm. When a Byzantine activity is detected by a correct process, the correct process notifies other processes, essentially, invoking the slow algorithm.

**Preliminaries.** In our schemes, a process  $p$  is replaced whenever enough testimonies of a two-faced behavior of  $p$  are collected. When there is no sufficient number of testimonies for a two-faced behavior of a particular process  $p$ , several replacements may take place. A replacement may be scheduled for the processes that provide the testimonies (possibly as a sacrifice action), and also for the process that is blamed to be two-faced.

A *consensus vector* of  $n$  inputs is required for maintaining the replicated state machine as described, for example in [3]. We are interested in providing a *consensus vector* solution with Byzantine detection, as we define next. The properties of the consensus vector task are defined in terms of  $n$  inputs. In a more formal way:

- **Agreement.** All non-faulty processes agreed on the same vector. There are not two non-faulty processes that decide on different vectors.
- **Validity.** Let  $V$  be the decision vector.  $\forall i \in N$ , if  $p_i$  is correct then  $vector[i]$  is the input value of  $p_i$ .

As for the detection properties, the following are defined:

- **Completeness.** A process  $p_i$  acting in a two-faced manner that causes the system not to decide in a *fast termination* fashion is suspected by some correct process  $p_j$  (and eventually replaced by the hypervisor).
- **Sacrifice.** If process  $p_i$  is suspected by at least  $t + 1$  processes,  $p_i$  is the only one to be replaced. Otherwise,  $p_i$  and the processes that suspect  $p_i$  are replaced.

**The Fast Algorithm** (Algorithm 1). The fast algorithm takes place in the first three rounds following a consensus invocation. In the first round (lines 1–2), each process sends its input value while in the second round (lines 3–5) each process sends the values it received in the previous round. After these two rounds, each process looks for conflicting messages as an indication for Byzantine activity and extracts a suspect list. In the third round, a process sends a bit (an indication bit) with value of 1 (*suspect message*) as an indication for Byzantine activity if such activity is detected. Then, processes start a new binary consensus invocation (phase 2 line 9). The input value is determined as follows: if a process receives suspect messages, the process uses 1 as an input value and uses 0 otherwise.

Either way, starting from this point, the processes make a speculative execution of the consensus. The processes use the consensus vector of the first two rounds as a decision value, save the state before the consensus execution and concurrently, start the consensus and a new fast algorithm. If the decision

value is 0, then the speculative execution appears to be the right execution and the saved state is discarded. Otherwise, the decision value equals 1, meaning a Byzantine activity occurred. The processes roll back to the state before making the consensus and start the slow algorithm that will replace the Byzantine processes that cause disagreement on the input vectors. The relevant suspect lists that have been recorded in the corresponding incarnation of the fast algorithm is used by the slow one.

An *early stopping* Byzantine agreement is used to agree on the indication bit. For example, one may use the algorithm suggested in [?,4]. Such an *early stopping* algorithm terminates in  $\min\{t+1, f+1\}$  rounds, where  $t$  is the maximum number of Byzantine and  $f$  is the actual Byzantine processes.

---

**Algorithm 1** Fast Algorithm for process  $p_i$ , Denote by  $N$  the group of processes' identities, by  $v_i$ , the input value of process  $p_i$ :

---

**Process fields** (initialized with each invocation):

$vector = [\perp, \dots, \perp]$   
 $vectors = [\perp, \dots, \perp]$   
 $slow = 0$ ,  $suspects = \emptyset$

**Phase 1:**

**Round 1:**

- 1:  $\forall j \in N$ : send  $v_i$  to  $p_j$
- 2: on **receive** of  $v_j$  from  $p_j$ :  $vector[j] = v_j$

**Round 2:**

- 3:  $\forall j \in N$ : send  $vector$  to  $p_j$
- 4: on **receive** of  $vector_j$  from  $p_j$ :  $vectors[j] = vector_j$   
 $\triangleright$  Concurrently decides  $vector$  and speculatively executes the rest

**Round 3:**

- 5:  $suspects = getSuspects(vectors)$   $\triangleright$  check for conflicts
- 6: **if** ( $|suspects| \geq 1$ ) **then**  $\forall j \in N$ : send '1' to  $p_j$   $\triangleright$  indication for suspicion
- 7: on **receive** of '1' from  $p_j$ :  $slow = 1$

**Phase 2:**

- 8: **if**  $BA.decide(slow) = 1$  **then**  $\triangleright$  BA - optimal early stopping Agreement
  - 9: role back state and apply *SLOW algorithm* with  $suspects$  list
- 

The *getSuspected* function is used in the third round of the first phase. This function is used to detect Byzantine processes based on the received messages of the first two rounds. The function input is  $vectors_i$ , a vector of vectors that represents all the received messages of the two rounds. The value of  $vectors_i[k][j]$ ,  $k, j \in N$ , represents the value of  $p_j$  as received by  $p_k$  in the first round. The value of  $vectors_i[k][j]$  has been sent to  $p_i$  by  $p_k$  during the second round (if  $k = i$  then  $vectors_i[i][j]$  is known after the first round).

At the beginning, the function finds Byzantine processes and removes their values from  $vectors_i$ . Then the function searches for suspected processes. By analysing the vector it is possible to use a voting approach for each value pro-

posed on the basis of the value received during the exchanges in the first two rounds. Majority of process  $p_j$  is the most frequent value in  $\{vectors[i][j] : i \in N\}$ , where  $N$  is the unique processes' identifier.

In order to find faulty processes based on  $vectors_i$ , the *getSuspects* function checks, for each process  $p_k$ , if there are at least  $n - t$  processes that sent the same value in the second round, otherwise  $p_k$  is faulty. This check is done by examining the  $k$ -th entry of each vector in  $vectors_i$ . Then, the function counts how many times  $p_k$  provides a value  $v$  for  $p_j$  (given by  $vectors_i[k][j]$ ), such that  $v$  and the majority value for  $p_j$  are different. If it counts more than  $t$  occurrences,  $p_k$  is faulty. The function then checks, that the value received from  $p_k$  in the first round equals the majority value for  $p_k$ . After finding faulty process (or processes), their value is removed from  $vectors_i$ . However, when at least one faulty process had been found, less than  $t + 1$  testimonies are required to discover additional faulty processes. Finally, when no more faulty processes can be discovered, the remaining processes' conflicts are considered to be the *suspected* processes only. The function returns a union of the faulty and suspect groups.

---

**Algorithm 2** Description of the function **getSuspects**(vectors) for process  $p_i$

---

```

1: procedure getSuspects(vectors)
2:   maxFaults =  $t$ 
3:   faulty =  $\emptyset$ 
4:   repeat
5:     newFaulty =  $\emptyset$ 
6:     for  $k \in N \setminus \textit{faulty}$  do
7:       majorityDiff =  $\{j \in N \setminus \textit{faulty} : vectors[k][j] \neq \textit{majority}(j)\}$ 
8:       if ( $\textit{majority}(k) = \perp$ ) OR ( $vectors[i][k] \neq \textit{majority}(k)$ )
          OR ( $|\textit{majorityDiff}| \geq \textit{maxFaults} + 1$ ) then
9:         newFaulty = newFaulty  $\cup \{k\}$ 
10:      for  $k \in \textit{newFaulty}$  do
11:         $vectors[k] = [\perp, \dots, \perp]$ 
12:         $\forall j \in N : vectors[j][k] = \perp$ 
13:        maxFaults =  $t - |\textit{faulty}|$ 
14:        faulty = faulty  $\cup \textit{newFaulty}$ 
15:      until (newFaulty =  $\emptyset$  OR  $|\textit{faulty}| = t$ )
16:      suspects =  $\{k : k \in \{i, j\} \text{ s.t. } \exists j, i \in N \setminus \textit{faulty} \text{ } vectors[j][i] \neq \textit{majority}(i)\}$ 
17:      return faulty  $\cup \textit{suspects}$ 

18: procedure majority( $k$ )
19:   if  $vectors[i][k] = \perp$  then
20:     return  $\perp$ 
21:   return  $v$  s.t.  $|\{j \in N : vectors[j][k] = v\}| \geq n - t, \perp$  otherwise

```

---

The first two rounds of the algorithm are based on the exponential information gathering (EIG) Byzantine agreement algorithm introduced in [2]. In the



first round, each process sends its own input value. In round  $r > 1$ , each process sends all the messages it receives in round  $r - 1$ .

The following definitions and proofs are used to explain the detection process of the fast algorithm. The main detection process is done by the function *getSuspects* (Algorithm 2) after collecting messages of two rounds. The function starts with identification of the Byzantine processes, then the function finds suspect processes using  $p_i$  messages (i.e., from  $p_i$ 's point of view). The definitions *c-order lie with respect to  $p_i$*  and *A c-discoverable with respect to  $p_i$*  are used to explain the detection of Byzantine process and the definition  *$p_i$  co-suspects  $p_j$  and  $p_k$*  is used for explaining the detection of suspected processes. Then lemmas are used, based on those definitions, to prove the correctness of the detection algorithm. First, *getSuspects* function tries to identify Byzantine processes by finding *A c-discoverable with respect to  $p_i$*  processes. Then, identifies the suspects process by looking for processes  $p_j, p_k$  such that  *$p_i$  co-suspects  $p_j$  and  $p_k$* .

**Definition 1.**  *$p_i$  co-suspects  $p_j$  and  $p_k$ .* Let  $r > 1$ , process  $p_i$  co-suspects  $p_j$  and  $p_k$  if in round  $r$ ,  $p_j$  has sent to  $p_i$  value  $v$ , which supposed to be the value that  $p_k$  sent in round  $r - 1$ , while a majority of processes have sent value  $v' \neq v$  to  $p_i$  as the value sent by  $p_k$  in round  $r - 1$ .

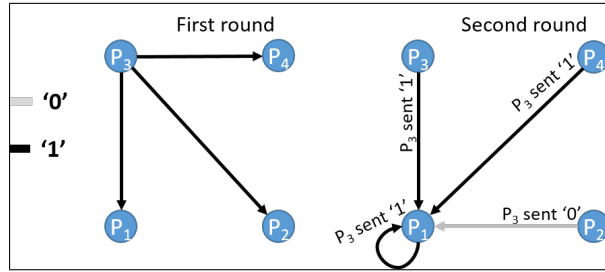
**Definition 2.** *c-order lie with respect to  $p_i$ .* Let  $c \geq 1$ ,  $C$  - a correct processes group of size  $c$ , called the lied group,  $p_i \in C$ . There are two kinds of c-order lie:

- **Two-faced.** Process  $p_j, j \in N/C$  sends value  $v$  to all processes in  $C$  and a different value (or values) to others not in  $C$ .
- **Anomalous.**  $\forall p_i \in C$ ,  $p_i$  co-suspects  $p_j$  and  $p_k$ ,  $k \in N$

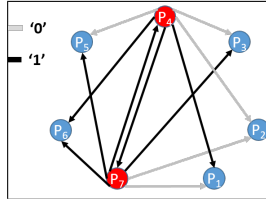
A better understanding of the difference between two-faced and anomalous requires a look in Algorithm 1. In the first round only one value, the input value, is sent. In the second round,  $n$  values are sent (as a single message). A two-faced lie can be made in the first round and be discovered in the second round. In that case, in the first round, the process may send  $v$  as its initial value to  $0 < c < n$  processes and value  $v'$ ,  $v \neq v'$ , to others, but it can only lie concerning a single value in its message. On the other hand, an anomalous lie can be made in the second round, where there are  $n$  possibilities to lie for each one of the  $n$  values.

*c-order lie* leads to the *c-discoverable* definition. Usually, given  $t + 1$  testimonies for process  $p$  to be Byzantine, process  $p$  is doomed to be Byzantine. However, if we already count  $x$  Byzantine processes, then only  $t + 1 - x$  testimonies are required to discover another Byzantine process. Using this fact, the algorithm starts looking for Byzantine processes using  $t + 1$  testimonies to find  $x$  Byzantine processes, then uses  $t - x + 1$  testimonies, and so on, as long as new Byzantine processes are found. Note, that both lies are two-faced, but the way of detecting those lies is different.

**Definition 3.** *A c-discoverable with respect to  $p_i$  process.* Let  $1 \leq c \leq t + 1$ , a process  $p_j$  is c-discoverable with respect to  $p_i$  if  $i \neq j$ , and  $p_j$  made a c-order lie to the lied group  $C$ ,  $p_i \in C$  and:



**Fig. 1.** We focus only on messages regarding process  $p_3$ .  $p_1$  co-suspects  $p_2$  and  $p_3$ . Three processes (including  $p_1$ ) notify  $p_1$  that  $p_3$  sent 1 at the first round and  $p_2$  tells  $p_1$  that  $p_3$  sent 0 at the first round.



**Fig. 2.**  $p_4$  sends different values to different processes.  $p_4$  is 3-discoverable ( $C = \{5, 3, 2\}$ ) and  $p_7$  is 2-discoverable ( $C = \{1, 2\}$ ).  $p_4$  and  $p_7$  will cause the processes to run the slow algorithm.  $p_4$  and  $p_7$  will be replaced by the hypervisor then.

- $c = t + 1$  or
- $c < t + 1$ , and there is a group  $G = \{k \in N : p_k \text{ is a } c'\text{-discoverable with respect to } p_i, c' > c\}$ , such that  $|G| > t - c$

In Figure 1, we focused only on messages involving  $p_3$ , even though other messages are sent as well. In the first round  $p_1$ ,  $p_2$  and  $p_4$  receive the value 1 from  $p_3$ . In the second round,  $p_4$  and  $p_1$  notify  $p_1$  ( $p_1$  sends the message to itself), that  $p_3$  sent 1 in the first round, while  $p_2$  claims that the value 0 had been sent by  $p_3$ . In that case,  $p_1$  has two processes, including itself, that report 1, and one process that reports 0.  $p_1$  cannot identify whether  $p_3$  or  $p_2$  are faulty and  $p_1$  suspects them both.  $p_1$  co-suspects  $p_2$  and  $p_3$ . Figure 2 depicts the case in which  $p_4$  and  $p_7$  are 4 and 3-discoverable, respectively. The algorithm first detects  $p_4$  and only then can detect  $p_3$ .

**Lemma 1.** *If  $p$  is a  $c$ -discoverable with respect to  $p_i$ , then  $p$  is detected as Byzantine by  $p_i$ .*

*Proof.* The proof is given by decreasing induction on the value of  $c$ ,  $1 \leq c \leq t + 1$ , starting with  $c = t + 1$  as the base case. Let  $c = t + 1$ ,  $p_i$  has  $t + 1$  testimonies claiming  $p_j$  as faulty. Since there are at most  $t$  faulty processes, at least one of them is correct. Thus  $p_j$  is faulty.

The inductive step. Assuming the claim is correct for  $c < t + 1$ , we show its correctness for  $c' = c - 1$ . By definition, there is a group of processes  $G$  wherein, each process is  $c'$ -discoverable with respect to  $p_j$ , such that  $c' > c'$ . Therefore, by the induction assumption, all processes in  $G$  will be detected as faulty by  $p_i$ .  $|G| > t - c'$  means that  $p_i$  has already discovered at least  $t - c'$  faulty processes. By definition of  $c$ -order lie with respect to  $p_i$ ,  $c$  refers to the number of correct processes that have been lied to. Consider the case in which  $N' = N/G$ , in such a case any group of  $c'$  processes in  $N'$  includes at least one correct process. Now, given  $c'$  testimonies claiming  $p_j$  as faulty, there is at least one correct process, therefore  $p_j$  is faulty and discovered by  $p_i$ .

The following lemmas can be used for the exponential information gathering (EIG) Byzantine agreement algorithm [2] where the algorithm terminates in  $t + 1$  rounds. In our algorithm, these lemmas hold for the first two rounds of the fast algorithm.

**Lemma 2.** *Let  $p_j$  be a  $c$ -discoverable with respect to  $p_i$  in round  $r \geq 1$ . If  $p_j$  is two-faced, assuming at least one round left,  $p_i$  detects  $p_j$  as faulty in round  $r + 1$ .*

*Proof.* Let  $p_j$  be a  $c$ -discoverable with respect to  $p_i$  in round  $r$  with a two-faced lie. It follows that  $p_j$  sends a value  $v$  to group  $C$  of  $c$  correct processes while sending a different value (or values) to other correct processes. Processes in  $C$  will identify the different value sent to the other processes only in the next round when all the processes send their receiving value of the previous round.

**Lemma 3.** *Let  $p_j$  be a  $c$ -discoverable with respect to  $p_i$  in round  $r \geq 1$ . If  $r > 1$  and  $p_j$  is anomalous,  $p_i$  detects  $p_j$  as faulty in round  $r$ .*

*Proof.* Let process  $p_j$  be a  $c$ -discoverable with respect to  $p_i$  in round  $r > 1$  with an anomalous lie. It follows that  $p_j$  claims that it received certain messages from processes in  $C$  (of  $c$  correct processes), while these messages are different from the messages received by  $p_i$  from the processes in  $C$ . Since processes in  $C$  are correct, they sent the same message to all processes, including the  $n - t$  correct processes. Let  $p_k \in C$ , when  $p_i$  checks the received messages from all processes for  $p_k$ ,  $p_i$  will identify at least  $n - t$  messages carrying the same value as  $p_i$  got last round from  $p_k$ . Giving the fact that  $p_j$  is  $c$ -discoverable with respect to  $p_i$ ,  $p_i$  has  $c$  testimonies claiming  $p_j$  for being faulty. Thus,  $p_i$  can detect  $p_j$  as faulty.

*Remark 1.* Lines 5–9 in Algorithm 2 are searching for two-faced or anomalous processes with  $maxFaults$  testimonies.

*Remark 2.* Line 16 in Algorithm 2 when applied with  $vectors_i$  is searching for  $p_j, p_k$  such that  $p_i$  co-suspects  $p_j$  and  $p_k$ .

**Lemma 4.** *Let  $p_j$  a  $c$ -discoverable with respect to  $p_i$ , then  $getSuspects(vectors_i)$  for  $p_i$  will return  $p_j$  as faulty.*

*Proof.* Let  $p_j$  be a  $c$ -discoverable with respect to  $p_i$ , then there is a group  $G = \{k \in N : p_k \text{ is a } c'\text{-discoverable with respect to } p_i, c' > c\}$ , such that  $|G| > t - c$ . Lines 5–9 of *getSuspects* (Algorithm 2) search for  $(\text{maxFaults} + 1)$  testimonies of *two-faced* or *anomalous* for each process.

The proof is given by decreasing induction on the value of  $c$ ,  $1 \leq c \leq t + 1$ , starting with  $c = t + 1$  as the base case. Let  $c = t + 1$ ,  $p_i$  has  $t + 1$  testimonies claiming  $p_j$  as faulty. In line 2,  $\text{maxFaults}$  is initialized to  $t$ , then lines 5–9 searching for  $t + 1$  testimonies of *two-faced* or *anomalous*. A  $(t + 1)$ -discoverable with respect to  $p_i$  process is discovered after executing lines 5–9.

After discovering a group of  $c$ -discoverable with respect to  $p_i$ ,  $\text{maxFaults}$  is decreased by the size of this group (line 13).

The inductive step. Assuming the claim is correct for  $c < t + 1$ , we show its correctness for  $c' = c - 1$ . By definition, there is a group of processes  $G$  wherein, each process is  $c''$ -discoverable with respect to  $p_j$ , such that  $c'' > c'$ . Therefore, by the induction assumption, all processes in  $G$  will be detected as faulty by  $p_i$ .  $|G| > t - c'$  means that  $p_i$  has already discovered at least  $t - c'$  faulty processes. Thus,  $\text{maxFault}$  eventually is updated to be  $t - |G|$  and lines 5–9 will search for at least  $\text{maxFault} + 1$  ( $t - c' + 1$ ) testimonies and  $p_j$  will be discovered.

**Lemma 5.** *Let  $p_j$  be a  $c$ -discoverable with respect to  $p_i$ , with *two-faced* in round 1 or *anomalous* in round 2, where  $p_i$  is a correct process. Then all processes moved to the *slow algorithm*, where at least one process has non-empty suspect list.*

*Proof.* Let  $p_j$  be a  $c$ -discoverable with respect to  $p_i$ , by Lemma 4  $p_i$  will discover  $p_j$  by round 3 and will send indication bit '1' to all processes. Then in phase 3 a Byzantine agreement will be executed. Since  $p_i$  is correct, all correct processes' input value will be '1', the decision value will be '1' and all correct processes will move to the *slow algorithm*.

*Claim.* After applying *getSuspects*( $\text{vector}_i$ ) for each correct process  $p_i$  the following holds:

- Let  $p_j$  be a Byzantine process that exhibits Byzantine behavior. If  $p_j$  is a  $c$ -discoverable with respect to  $p_i$ ,  $p_j$  will be included in  $p_i$ 's faulty list, where  $p_i$  is a correct process. Otherwise, if  $p_i$  co-suspects  $p_j$  and  $p_k, k \in N$ ,  $p_j$  will be included in  $p_i$ 's suspect list, where  $p_i$  is correct process.

*Proof.* Suppose  $p_j$  acts in a Byzantine two-faced fashion in the first two round of the *fast algorithm*. In case  $p_j$  acts in a  $c$ -discoverable fashion, then by Lemma 4  $p_j$  will be discovered as faulty. Otherwise, suppose  $p_i$  co-suspects  $p_j$  and  $p_k, k \in N$ , by Remark 2 it will be discovered as suspected.

**Lemma 6.** *If there is no Byzantine activity, while running the *fast algorithm*, the algorithm satisfies Agreement, validity, and termination.*

*Proof.* Suppose there is no Byzantine activity while running the *fast algorithm*. *Agreement.* Each correct processes received the same messages for creating  $\text{vector}_i$ .

Hence, they agreed on the same vector.

*Validity.* Since no Byzantine activity done, the value at entry  $i$  of the vector (aka *vector*[ $i$ ]) is the input value of  $p_i$ .

*Termination.* The algorithm composed of two phases. The first phase (lines 1–7) contains three rounds and the second phase (lines 8–11) composed of Byzantine agreement execution (line 8), which is finite due to the Byzantine agreement termination property, then the algorithm terminates. After termination, the slow algorithm might be applied (a separate proof for the slow algorithm in the sequel).

**Theorem 1.** *The fast algorithm satisfies completeness.*

*Proof.* Suppose there is a Byzantine behavior of  $c$ -order lie (either two-faces or anomalous) occurs by process  $p_k$  while running the Fast algorithm. suppose the Byzantine activity occurs at phase 1, by Claim 4 there is a correct process  $p_i$  that the return value of applying *getSuspects* function will return process  $p_k$  either as faulty or suspect. Thus,  $p_i$  will let the other correct processes know about the Byzantine activity and the Slow algorithm will be scheduled. Otherwise, suppose that Byzantine activity occurs at phase 2 only, by the speculative execution method, the fast algorithm keeps running and if the decision value equals 1, a rollback is scheduled, and the slow algorithm is scheduled to replace the faulty process that has been detected before running the consensus.

*Sacrifice* property will be dealt with later in the Slow algorithm.

## 5 Using Byzantine Agreement Objects

### 5.1 Global Hypervisor

**The Slow Algorithm** (Algorithm 3). The algorithm consists of  $n$  stages that run one after another, one for each process. The stage computation steps are presented in Algorithm 3 and composed of 3 phases. The first 2 phases are done by the processes, and the third phase is done by the *global hypervisor*, a single hypervisor for controlling all the processes as defined earlier. In stage  $s$ , phase 1 (lines 1–7), process  $p_i$ , such that  $(s \bmod n) + 1 = i$ , is the sender. The sender sends its input value,  $v_i$ , to all the other processes. Then the processes invoke an initialized version of a Byzantine consensus on  $v_i$ . Process  $p_j$  sets the *suspect* variable to 1 (line 7) if an agreement is reached, but the decision value is differed from the received value, or  $p_j$  had suspect  $p_i$  in the *fast algorithm* already. In that cases,  $p_j$  suspects  $p_i$ . In phase 2 (lines 8–10) each process, with *suspect* variable set to 1, sends a suspect message to the *global hypervisor* (line 10).

The hypervisor can decide to eliminate or replace a process based on the testimonies of other processes. In phase 3 (lines 11–14), if the hypervisor receives at least  $t + 1$  testimonies claiming that process  $p_i$  is faulty,  $p_i$  is doomed to be a Byzantine process since at least one non-faulty process’s testimony exists. If the number of testimonies is less than  $t + 1$ , it is unclear whether  $p_i$  is Byzantine or

---

**Algorithm 3** Code for process  $p_i$  and hypervisor in stage  $s$ , each process starts with suspect list it discovered in the fast algorithm:

---

**Process fields:**  $in_i[1 \dots n]$  initially [null, ..., null],  
 $suspect = 0$   
 $suspects$  = initialized from *fast algorithm*  
 $BA[1 \dots n]$  initially [null, ..., null]

**Phase 1 of stage  $s$ :**

```

1:   if  $s \pmod n + 1 = i$  then
2:     send  $v_i$  to all processes
3:   else                                     ▷ upon receiving  $v_j$  from  $p_j$ 
4:      $in_i[j] = v_j$ 
5:      $dec_i[j] = BA[j].decide(v_j)$          ▷ BA - Byzantine Agreement object
6:     if  $dec_i[j] \neq in_i[j]$  OR  $j \in suspects$  then
7:       turn on hypervisor ;  $suspect = 1$ 

```

**Phase 2 of stage  $s$ :**

```

8:   if  $s \pmod n + 1 \neq i$  AND  $suspect = 1$  then
9:      $j = s \pmod n + 1$ 
10:    send  $suspect(j)$  to hypervisor

```

---

Hypervisor code:

---

**Phase 3 of stage  $s$ :** ▷ Let  $P = \{i_1, \dots, i_k\}$  group of processes that send  $suspect(i)$  messages.

```

11:  if  $(k > t)$  then
12:    replace  $p_i$ 
13:  else
14:     $\forall j \in P \cup \{i\}$ : replace  $p_j$ 

```

---

not. Either the testimonies are correct and  $p_i$  is Byzantine, or the  $t$  Byzantine processes worked together to incriminate  $p_i$ . Thus, in this case, the hypervisor has to replace them all, i.e.,  $p_i$  and the other processes that sent the testimonies.

By this approach, once Byzantine activity occurs by sender process  $p_i$ , it will be replaced by the hypervisor. Still, sometimes some correct processes would be replaced along with  $p_i$ . As a conclusion, the addition of third party (the hypervisor) cannot identify Byzantine process  $p_i$  unless there are at least  $t + 1$  processes that claim  $p_i$  is Byzantine. Otherwise, in the worst case scenario, the only possibility left is to suspect all the  $t + 1$  processes. That way, in all cases Byzantine process  $p_i$  that has been discovered in the fast part is doomed to be replaced at stage  $s'$ , such that  $(s' \pmod n) + 1 = i$ . At the end of the  $n$  stages, each Byzantine process that has been discovered in the fast algorithm will be replaced and each non-faulty process will have the same vector of values. The next theorem summarizes the above observations.

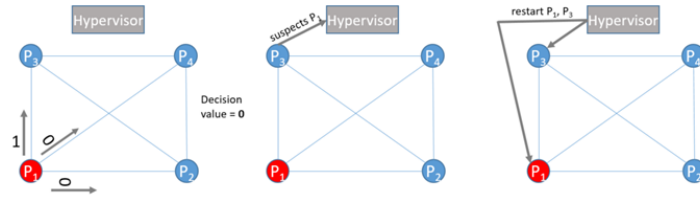
**Theorem 2.** *Algorithm 3 satisfies agreement, validity, and termination.*

*Proof. Agreement.* The algorithm consists of  $n$  stages, at each stage, all the correct processes agree on the same value for process  $p_i$  with  $s(\bmod n) + 1 = i$  through a Byzantine consensus (line 5). Eventually, after  $n$  stages, all correct processes received the same  $n$  values representing the vector of size  $n$ .

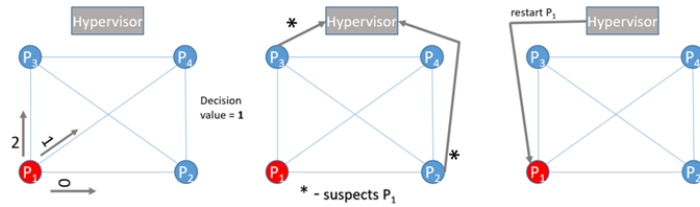
**Validity.** For each entry  $i$  in the vector a consensus had been made at stage  $s$ , such that  $i(\bmod n) + 1 = i$ . *Validity*, by definition, concerns only the input value of correct processes. If the sender is correct, it sent the same value,  $v$  to all processes in phase 1 (line 2). Then Byzantine agreement is invoked and all correct processes hold the same input value  $v$ . By validity property of Byzantine consensus, the decision value is  $v$ .

**Termination.** The algorithms consist of  $n$  stages. Each stage composed of three phases. The first phase contains one round and Byzantine agreement execution (lines 1–7), which is finite due to the Byzantine agreement termination property. The second phase lasts for one round (lines 8 – 10) and finally, the third phase (lines 11 – 14) contains the hypervisor part which requires one round. Given that stage is finite,  $n$  stages are also finite, thus the algorithm terminates.

Figures 3 to 6 describe the various scenarios for the case of  $n = 4, t = 1$ .



**Fig. 3.**  $P_1$  sends wrong value to  $P_3$ .  $P_3$  suspects  $P_1$  and both replaced by the hypervisor.



**Fig. 4.**  $P_1$  sends different values such that  $P_2, P_3$  suspect  $P_1$ , and  $P_1$  replaced by the hypervisor.

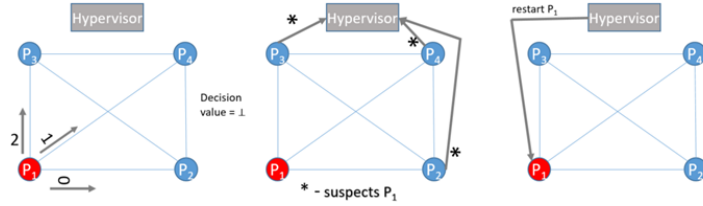


Fig. 5.  $P_1$  sends different values such that there is no agreement, and  $P_1$  replaced by the hypervisor.

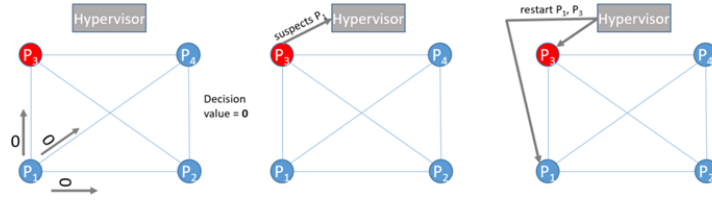


Fig. 6.  $P_1$  non-faulty, but incriminated by  $P_3$ . Both replaced by the hypervisor.

### 5.2 Local Hypervisor

In this part, we assume a different hypervisor, namely, a *local hypervisor* for each process. The *local hypervisor* can be turned on by its process and assumed to be correct all the time. The local hypervisors can send and receive suspects messages (as explained in the sequel) but cannot send messages to the processes. The *hypervisor* can replace the process based on the suspect messages.

**Algorithm 4.** The algorithm starts the same as Algorithm 3, in phase 1 there is a sender process sending its own input value to all processes, following by Byzantine consensus invocation for the received value, and suspect if the decision value is different from the received value or if the sender had already suspected in the *fast algorithm*. Algorithm 4 differs from Algorithm 3 following the detection part. A process that suspects the sender turns on its local hypervisor as an indication for the detection. In phase 2, the *local hypervisor* sends suspect messages to all other hypervisors. During phase 3, after all suspects messages of phase 2 have been received, the local hypervisor that controls the sender replaces the sender as a consequence of receiving at least one suspect message. The local hypervisor of the other processes, different from the sender, replaces their hosted process only when there are less than  $t + 1$  suspect messages.

**Theorem 3.** *Algorithm 4 satisfies agreement, validity, and termination.*

*Proof. Agreement.* The algorithm consists of  $n$  stages, at each stage, all the correct processes agree on the same value for process  $p_i$  with  $s(\text{mod } n) + 1 = i$  through a Byzantine consensus (line 5). Eventually, after  $n$  stages, all correct processes receive the same  $n$  values representing the vector of size  $n$ .



---

**Algorithm 4** Code for separated hypervisors and process  $p_i$  in stage  $s$ , each process starts with suspect list it discovered in the fast algorithm:

---

*local hypervisor* turned off except for  $p_i$ , such that  $s \pmod n + 1 = i$

**Process fields:**

$in_i[1 \dots n]$  initially [null, ..., null],  
 $BA[1 \dots n]$  initially [null, ..., null]  $\triangleright$  BA - Byzantine Agreement object  
 $suspects$  = initialized from *fast algorithm*

**Phase 1 of stage  $s$ :**

```

1:   if  $s \pmod n + 1 = i$  then
2:     send  $val(v_i)$  to all processes
3:   else  $\triangleright$  upon receiving  $v_j$  from  $p_j$ 
4:      $in_i[j] = v_j$ 
5:      $dec_i[j] = BA[j].decide(v_j)$   $\triangleright$  BA - Byzantine Agreement object
6:     if  $dec_i[j] \neq in_i[j]$  OR  $j \in suspects$  then
7:       turn on local hypervisor

```

---

Hypervisor code (for  $p_i$ 's turned on (active) local hypervisor):

---

**Phase 2 of stage  $s$ :**

```

8:   if  $s \pmod n + 1 \neq i$  then
9:      $j = s \pmod n + 1$ 
10:    send  $suspect(j)$  to all active hypervisors
11:  else  $\triangleright$  counter-testimony for all active hypervisors
12:     $\forall j \in N$  s.t.  $p_j$ 's hypervisor is active: send  $suspect(j)$  to  $p_j$ 's local hypervisor

```

**Phase 3 of stage  $s$ :**  $\triangleright k$  - amount of received  $suspect(i)$  messages.

```

13:  if  $s \pmod n + 1 = i$  AND  $k > 0$  then
14:    replace  $p_i$ 
15:  else if  $k \leq t$  then  $\triangleright$  upon receiving  $k$  suspect(j) messages
16:    replace  $p_i$ 

```

---

**Validity.** For each entry  $i$  in the vector a consensus had been made at stage  $s$ , such that  $i \pmod n + 1 = i$ . *Validity*, by definition, concerns only the input value of correct processes. If the sender is correct, it sent the same value,  $v$ , to all processes in phase 1 (line 2). Then Byzantine agreement is invoked and all correct processes hold the same input value  $v$ . By validity property of Byzantine consensus, the decision value is  $v$ .

**Termination.** The algorithms consist of  $n$  stages. Each stage composed of three phases. The first phase contains one round and Byzantine agreement execution (lines 1–7), which is finite due to the Byzantine agreement termination property. The second phase lasts for one round (lines 8–12) and finally, the third phase (lines 13–16) which requires one round. Given that stage is finite,  $n$  stages are also finite, thus the algorithm terminates.

**Theorem 4.** *The slow algorithm satisfies completeness and sacrifice.*

*Proof. Completeness.* As seen in the algorithm, a process can lie either as the sender (line 2) and make up to processes to be restarted, or as a receiver (line 7) and it will make itself and the sender be restarted. Either way the faulty process will pay for it.

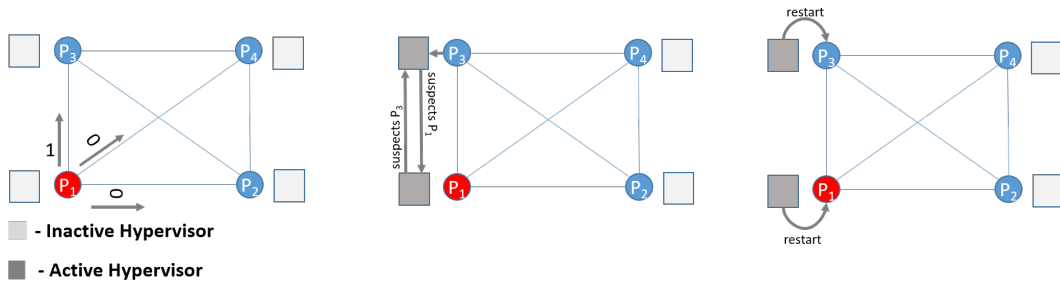
**Sacrifice.** Following lines 13 – 16, if there is not enough, e.g. at least  $t + 1$ , receivers that claims the sender (process  $i$  with  $s \pmod n + 1 = i$ ) for being faulty, the sender and the reporting receivers will be restarted as an act of sacrifice. Otherwise, the sender alone will be restarted.

**Theorem 5.** *The combinations of Algorithm 4 and Algorithm 1 satisfies Completeness and sacrifice.*

*Proof. Completeness.* As already mentions, the fast algorithm satisfy completeness (Theorem 1) by recording and delivering the faulty and suspects process list to the slow algorithm. Then the slow algorithm continues satisfies Completeness (Theorem 4) collect more evidences (if there is any) for Byzantine behavioral.

**Sacrifice.** Restart of processes is done only in the slow part. As seen earlier (Theorem 4), the slow algorithm satisfies sacrifice.

Figures 7 to 9 describe the various scenarios for the case of  $n = 4, t = 1$ .



**Fig. 7.**  $P_1$  sends wrong value to  $P_3$ .  $P_3$  suspects  $P_1$  and both replaced by their local hypervisors.

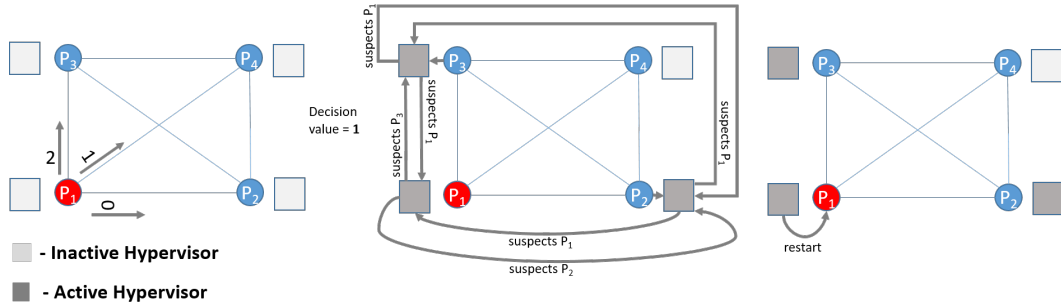


Fig. 8.  $P_1$  sends different values such that  $P_2, P_3$  suspect  $P_1$ , and  $P_1$  replaced by its local hypervisor.

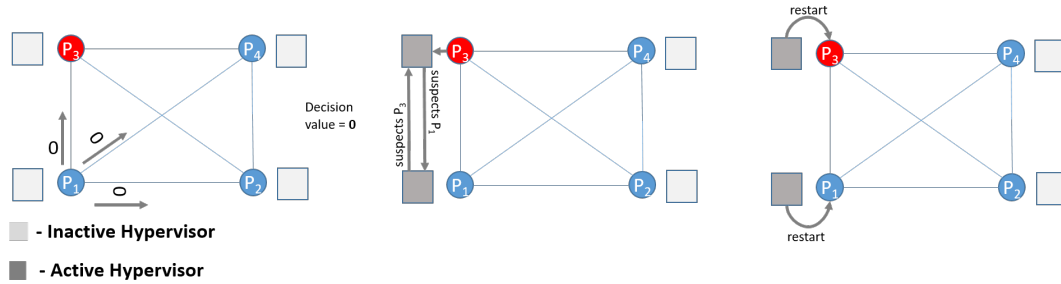


Fig. 9.  $P_1$  non-faulty, but incriminated by  $P_3$ . Both replaced by their local hypervisors.

## 6 Conclusion

This paper presented an approach to detect and remove Byzantine processes from consensus-based computation. Long-lived computation relying on consensus (e.g., state machine replication) may benefit from our solution as it allows to continuously monitor the computation and inhibits Byzantine processes to act, if they wants to remain in the computation.

## References

1. Abraham, I., Dolev, D. “Byzantine agreement with optimal early stopping, optimal resilience and polynomial complexity, *Proc. of the 47th Annual on Symposium on Theory of Computing, (STOC)*, 2015.
2. Bar-Noy, A., Dolev, D., Dwork, C., Strong, H.R. “Shifting gears: Changing algorithms on the fly to expedite byzantine agreement,” *Information and Computation* 97(2), 205-233 (1992).
3. Binun, A., Coupaye, T., Dolev, S., Kassi-Lahlou, M., Lacoste, M., Palesandro, A., Yagel, R., Yankulin, L., “Self-stabilizing byzantine-tolerant distributed replicated state machine,” *Proc. of the 18th International Symposium on Stabilization, Safety and Security of Distributed Systems, (SSS)* 2016.

4. Dolev, D., Reischuk, R., Strong, H.R., "Early stopping in byzantine agreement," *J. ACM* 37(4), 720-741 (Oct 1990)
5. Kung, H.T., Robinson, J.T., "On optimistic methods for concurrency control," *ACM Trans. Database Syst.*, 6(2), 213-226 (Jun 1981)
6. Mostfaoui, A., Raynal, M., "Intrusion-tolerant broadcast and agreement abstractions in the presence of byzantine processes," *IEEE Transactions on Parallel and Distributed Systems* 27(4), 1085-1098 (April 2016).
7. Pease, M., Shostak, R., Lamport, L., "Reaching agreement in the presence of faults," *J. ACM* 27(2), 228-234 (Apr 1980).