Byzantine Reliable Broadcast

An Adaptive Protocol for the Fault Detection in the Authenticated Double-Echo Broadcast

Giuseppe Daidone 2122594

February 2024

1 Introduction

Distributed Broadcast Communication can be affected by Byzantine processes and classical broadcast algorithms may not tolerate them. There exist Byzantine Tolerant Broadcast Algorithms which are able to deal with Byzantine processes (e.g. Authenticated Echo Broadcast, Authenticated Double-Echo Broadcast), but they work only with the strong assumption that every process knows how many Byzantine processes are present in the system at the start execution of the algorithm and never change their number. This study aims to relax the previous assumptions and develop a modified version of a broadcast protocol which would be adaptive to the number of Byzantine processes.

2 Definitions

Given a Distributed System composed by the process set $\Pi = \{p_1, p_2, ..., p_N\}$, a Byzantine process $p_i \in \Pi$ is a process that can deviate arbitrarily from the instructions an algorithm assigns to them and it acts as if it were deliberately preventing the algorithm from reaching its goals.

The communication between processes is Authenticated and Reliable using cryptography. In this study it would be used the Authenticated Perfect Point-to-Point Link primitive guaranteeing: Reliable Delivery, No Duplication, Authenticity.

The reference protocol would be the Authenticated Double-Echo Broadcast[1], which would implement the Byzantine Reliable Broadcast primitive guaranteeing: Validity, No Duplication, Integrity, Consistency, Totality.

3 Overview of the Adaptiveness Problem

The Authenticated Double-Echo Broadcast Algorithm is defined below.

Algorithm 1 Authenticated Double-Echo Broadcast

Implements: ByzantineReliableBroadcast, **instance** *brb*, with sender *s*.
Uses: AuthPerfectPointToPointLinks, **instance** *al*.

```
upon event ⟨ brb, Init ⟩ do
    sentecho := FALSE;
    sentready := FALSE;
    delivered := FALSE;
    echos := [⊥]^N;
    readys := [⊥]^N;

upon event ⟨ brb, Broadcast | m⟩ do                    ▷ executed only by s
    forall q ∈ Π do
        trigger ⟨ al, Send | q, [SEND, m] ⟩;

upon event ⟨ al, Deliver | p, [SEND, m] ⟩ such that p = s and sentecho =
FALSE do
    sentecho := TRUE;
    forall q ∈ Π do
        trigger ⟨ al, Send | q, [ECHO, m] ⟩;

upon event ⟨ al, Deliver | p, [ECHO, m] ⟩ do
    if echos[p] = ⊥ then
        echos[p] := m;

upon exists m ≠ ⊥ such that #(p ∈ Π|echos[p] = m) > (N+f)/2 and
sentready = FALSE do
    sentready := TRUE;
    forall q ∈ Π do
        trigger ⟨ al, Send | q, READY, m ⟩;
```

upon event $\langle$ *al*, *Deliver* $\mid p$, [$\textsc{Ready}, m$] $\rangle$ **do**
  if $readys[p] = \perp$ **then**
    $readys[p] := m$;

upon exists $m \neq \perp$ **such that** $\#(p \in \Pi | readys[p] = m) > f$ **and** $sentready = \textsc{False}$ **do**
  $sentready := \textsc{True}$;
  forall $q \in \Pi$ **do**
    trigger $\langle$ *al*, *Send* $\mid q$, [$\textsc{Ready}, m$] $\rangle$;

upon exists $m \neq \perp$ **such that** $\#(p \in \Pi | readys[p] = m) > 2f$ **and** $delivered = \textsc{False}$ **do**
  $delivered := \textsc{True}$;
  trigger $\langle$ *brb*, *Deliver* $\mid s, m$ $\rangle$;

The variable f is representing the number of Byzantine processes in the system and the variable N represents the total number of processes in the system. Therefore, every process does not know which are the Byzantine processes, but knows exactly how many of them are running. The correctness is ensured if and only if $N > 3f$.

Moving in a real-case scenario, where processes may access to a global variable to get N, during the running of the algorithm one or more of them may be intercepted and controlled by an adversary that makes them behave as it wants. Now, processes must need to detect a Byzantine process correctly so that they can be adaptive and react to changes and increment the value f.

4 Byzantine Adaptive Solution

Assumptions:

- Process set Π with size of N.
- Synchronous System.
- Every process can access to a global variable that returns the set of processes into the system and their number.
- At the initial state, every process believes that there is no Byzantine process into the system $correct = \Pi$, $suspected = \emptyset$, $faulty = \emptyset$ and $f = 0$.

- Every process can communicate with every other process through Authenticated Perfect Point-to-Point Links (digitally signed messages).
- Byzantine can have access to the broadcast algorithm and it cannot interfere or modify the links level or below.
- Byzantine processes can behave as they want and it is not predictable the precise instant when it became faulty or if it was faulty since the initial state.
- Every process keeps track of the broadcast sender in a local variable s.

In order to define an adaptive solution, there is the need to study all different cases where Byzantine processes may act. The original protocol may be divided into four main phases:

- Send: a process, referred as sender, broadcast a message which is sent to all the processes into the system;
- Echo: each process does an echo of the message, replaying it to all the processes;
- Ready: each process that is ready to deliver, communicates the willing of delivering the message to all the processes.
- Deliver: each process delivers the message of the broadcast.

This scheme of execution may be broken because of Byzantine process in different ways, for example: not fully (or partially) participating in a phase of the protocol, send different messages to other processes misleading them in any phase of the protocol (even the broadcast original message), etc. When Byzantine is able to compromise the broadcast communication, the algorithm will terminate the execution for every process and every correct process cannot continue in further communication. The adaptive solution must correctly detect the number of Byzantine processes, but also it must identify them so that every correct process will not communicate with them anymore.

A correct process, after each Echo and Ready phase (the most crucial), must analyse the set of received messages, trying to find heterogeneity and estimate the parameter *f*. A Consistent Message is defined as the message more frequent excluding the sender, the number of them determine whether a process may take a decision about proceeding in the algorithm and if it is able to identify Byzantine processes. If a process receives a message m' from the sender in the execution of the algorithm, that differs from the broadcast message m, the process is able to conclude that the sender is Byzantine. When a process is sure about a faulty process (under certain assumptions of the system), then it communicates it with any other process into the system and, if the Byzantine is the sender, protocol stop its execution because the sender's correct behavior was compromised.

4.1 Echo Phase

Suppose that a process p_0 broadcast the message m and that the echo set of a certain process p_i is the following:

$$p_i : \{p_0 : m, p_1 : m, p_2 : m', p_i : m\}_{Echo}$$

In this set, the consistent message $cons = m$, because this message is the more frequent one:

$$p_1 \text{ and } p_i \text{ echo the message } m$$
$$p_2 \text{ echo the message } m'$$
$$\#cons = 2, \text{ over 3 processes excluding } p_0$$

so p_i could conclude that m is the correct broadcast message (true), but could it infer something about the correctness of p_2? The answer to this question is no, but for sure there is one Byzantine process, so $f_i = 1$. The are two different cases:

- p_0 may be Byzantine and misled p_2, so that it believes that the broadcast message is m', or
- p_2 is actually the Byzantine.

Since the process p_i is not sure which is the Byzantine, it needs to move to the next phase. In order to proceed, it must exclude from the communications p_2 and not the sender p_0. This suspicion may be confirmed in the next phase or it can revise its decision and correctly detect p_0. For the ECHO phase, p_i executes $suspected = suspected \cup \{p_0, p_2\}$. In the adaptive solution there is the need to trust the sender until the READY phase, referred to as initial trust of the sender.

Suppose now that the sender is the process which echo a different message, so p_i have the following echo set:

$$p_i : \{p_0 : m', p_1 : m, p_2 : m, p_i : m\}_{Echo}$$

p_i is able to correctly identify p_0 as the Byzantine process:

p_1, p_2 and p_i echo the message m

p_0 echo the message m', but p_i received m from p_0 in the SEND phase

so it detects it $faulty = faulty \cup \{p_0\}$ and declare $f_i = 1$.

However, a process is not able to identify the consistent message if the number of processes is not $N > 3f$. In the previous case there were $N = 4$ processes and $f = 1$ faulty; in addition, the previous condition is necessary but not sufficient to guarantee that every process proceeds in the execution. With the initial trust of the sender, a process could experience a symmetric situation where cannot infer anything and remains blocked. Consider the following example:

$$p_i : \{p_0 : m, p_1 : m', p_2 : m', p_i : m\}_{Echo}$$

In this set, a consistent message cannot be determined so p_i is blocked:

p_0 and p_i echo the message m

p_1 and p_2 echo the message m'

so p_i could guess the presence of Byzantine due to heterogeneity of the set, but cannot determine if:

- p_0 is Byzantine and misled p_1 and p_2, or
- p_0 is Byzantine and misled p_i, or
- p_1 and p_2 are Byzantine.

In addition, if p_i was misled by the sender p_0 but p_1 and p_2 were not, then p_1 and p_2 proceed with the algorithm execution suspecting p_i and p_0 of being the Byzantine, but excluding only p_i for the Ready phase due to the initial trust of the sender. In the next phase, every suspected process must be identified as faulty or recovered, so if p_1 and p_2 are correct they must study the behavior of the sender.

4.2 Ready Phase

Suppose that a process p_0 broadcast the message m and that the echo set of a certain process p_i is the following:

$$p_i : \{p_0 : m, p_1 : m, p_2 : m', p_i : m\}_{Echo}$$

p_i executes $suspected = suspected \cup \{p_0, p_2\}$ but the communication for the Ready phase continue with the set $\{sender\} \cup \{correct \backslash suspected\}$. Suppose that p_i construct the following ready set:

$$p_i : \{p_0 : m, p_1 : m, p_i : m\}_{Ready}$$

since this set is homogeneous p_i concludes:

- p_2 is a Byzantine process, $faulty = faulty \cup \{p_2\}$;
- $f_i = 1$;
- $correct = correct \cup \{sender\}$.

The conclusion can be made under the condition of $N > 3f$. Since p_i had the following message set received from p_0:

$$p_0 \text{ broadcast } m$$
$$p_0 \text{ echo } m$$
$$p_0 \text{ ready } m$$

p_i trust p_0 and p_2 is the only process to misbehave. Proving by contradiction, if p_2 was correct and was misled by p_0 in the Echo phase, then it would have the following set:

$$p_2 : \{p_0 : m', p_1 : m, p_2 : m, p_i : m\}_{Echo}$$

but this case was studied in the previous section, and p_2 is able to correctly identify p_0 as the Byzantine process because:

$$p_1,\ p_2 \text{ and } p_i \text{ echo the message } m$$
$$p_0 \text{ echo the message } m', \text{ but } p_2 \text{ received } m \text{ from } p_0 \text{ in the } \textsc{Send} \text{ phase}$$

so it detects it $faulty = faulty \cup \{p_0\}$, declare $f_2 = 1$ and protocol stops.

Suppose now the following echo sets:

$$p_1 : \{p_0 : m', p_1 : m', p_2 : m', p_i : m\}_{Echo}$$
$$p_2 : \{p_0 : m', p_1 : m', p_2 : m', p_i : m\}_{Echo}$$
$$p_i : \{p_0 : m, p_1 : m', p_2 : m', p_i : m\}_{Echo}$$

where p_1 and p_2 execute $suspected = suspected \cup \{p_0, p_i\}$, but p_i remains stuck due to the symmetry uncertainty. If they would have the following ready sets:

$$p_1 : \{p_0 : m', p_1 : m', p_2 : m'\}_{Ready}$$
$$p_2 : \{p_0 : m', p_1 : m', p_2 : m'\}_{Ready}$$

then, as the previous case, p_1 and p_2 conclude that:

- p_i is a Byzantine process, $faulty = faulty \cup \{p_i\}$;
- $f = 1$;
- $correct = correct \cup \{sender\}$.

However, if they would have different ready sets:

$$p_1 : \{p_0 : m', p_1 : m', p_2 : m'\}_{Ready}$$
$$p_2 : \{p_0 : m, p_1 : m', p_2 : m'\}_{Ready}$$

then, p_2 is able to correctly identify p_0 as Byzantine and protocol stops.

4.3 Byzantine Sender Detection Spreading

In this model of the protocol, whenever the source of broadcast is Byzantine then the algorithm would stop its execution. If this is the case, the protocol guarantees that at least one correct process detects the sender as faulty and it is not a suspected process for any other correct process, under the assumption of $N > 3f$. When a process detects the sender, it needs to spread this detection to all other processes that could be stuck or suspect (wrongly) other processes which are correct.

A solution for the Byzantine sender detection spreading is a special message ByzantineSender sent to each process, containing the echo set of process that detected the sender. The message could be sent after the Ready phase, when processes waits for a timer Δ before delivering. Consider the previous example of sender p_0 Byzantine and p_i stuck:

$$p_1 : \{p_0 : m', p_1 : m', p_2 : m', p_i : m\}_{Echo}$$
$$p_2 : \{p_0 : m', p_1 : m', p_2 : m', p_i : m\}_{Echo}$$
$$p_i : \{p_0 : m, p_1 : m', p_2 : m', p_i : m\}_{Echo}$$
$$p_1 : \{p_0 : m', p_1 : m', p_2 : m'\}_{Ready}$$
$$p_2 : \{p_0 : m, p_1 : m', p_2 : m'\}_{Ready}$$

p_1 and p_2 start the timer Δ. p_2 detects p_0 as Byzantine sender and executes:

$$suspected = suspected \setminus \{p_i\}$$
$$faulty = faulty \cup \{p_0\}$$
$$p_2 \text{ sends to } p_1 : [\text{ByzantineSender}, \{p_0 : m', p_1 : m', p_2 : m', p_i : m\}_{Echo}]$$
$$p_2 \text{ sends to } p_i : [\text{ByzantineSender}, \{p_0 : m', p_1 : m', p_2 : m', p_i : m\}_{Echo}]$$

p_1 is able to determine that p_i was misled by p_0 and executes:

$$suspected = suspected \setminus \{p_i\}$$
$$faulty = faulty \cup \{p_0\}$$
$$p_1 \text{ sends to } p_2 : [\text{ByzantineSender}, \{p_0 : m', p_1 : m', p_2 : m', p_i : m\}_{Echo}$$
$$p_1 \text{ sends to } p_i : [\text{ByzantineSender}, \{p_0 : m', p_1 : m', p_2 : m', p_i : m\}_{Echo}$$

Now p_i received two messages of ByzantineSender and, under the assumption of $N > 3f$ it can conclude:

- p_0 is a Byzantine process, $faulty = faulty \cup \{p_0\}$;
- $f = 1$.

4.4 Adaptive Algorithm

Algorithm 2 Adaptive Authenticated Double-Echo Broadcast

Implements: AdaptiveByzantineReliableBroadcast, **instance** *abrb*, with sender s.

Uses: AuthPerfectPointToPointLinks, **instance** *al*.

```
upon event ⟨ abrb, Init ⟩ do
    sentecho := FALSE;
    sentready := FALSE;
    delivered := FALSE;
    echos := [⊥]^N;
    readys := [⊥]^N;
    correct := Π;
    suspected := ∅;
    faulty := ∅;
    f := 0;
    byzantinesender := FALSE;
    senderechos := ∅;

upon event ⟨ abrb, Broadcast | m⟩ do                ▷ executed only by s
    forall q ∈ correct do
        trigger ⟨ al, Send | q, [SEND, m] ⟩;

upon event ⟨ al, Deliver | p, [SEND, m] ⟩ such that p = s and sentecho = FALSE do
    sentecho := TRUE;
    forall q ∈ correct do
        trigger ⟨ al, Send | q, [ECHO, m] ⟩;

upon event ⟨ al, Deliver | p, [ECHO, m] ⟩ do
    if echos[p] = ⊥ then
        echos[p] := m;
```

```
upon exists m ≠ ⊥ such that #(echos) ≥ (3·correct)/4 and Consistent(echos) = m do
    forall p ∈ echos do
        if echos[p] ≠ m and p ≠ s and p ∉ suspected do
            suspected := suspected ∪ {p};
            f := f + 1;
    if echos[s] ≠ echos[SELF] then
        faulty := faulty ∪ {s};
        correct := correct\{s};
        byzantinesender := TRUE;
        forall q ∈ correct do
            trigger ⟨ al, Send | q, [BYZANTINESENDER, echos] ⟩;
    elseif suspected ≠ ∅ and s ∉ suspected then
        suspected := suspected ∪ {s};

upon exists m ≠ ⊥ such that #(p ∈ correct|echos[p] = m) > (N+f)/2 and sentready = FALSE and byzantinesender = FALSE do
    sentready := TRUE;
    forall q ∈ {s} ∪ {correct\suspected} do
        trigger ⟨ al, Send | q, [READY, m] ⟩;

upon event ⟨ al, Deliver | p, [READY, m] ⟩ do
    if readys[p] = ⊥ and echos[p] = m then
        readys[p] := m;
    elseif readys[p] ≠ echos[p] and p ≠ s then
        f := f + 1;
        faulty := faulty ∪ {p};
        correct := correct\{p};
        suspected := suspected\{p};

upon #(readys) > 2f and byzantinesender = FALSE do
    if readys[s] ≠ echos[s] then
        f := f + 1;
        faulty := faulty ∪ {s};
        correct := correct\{s};
        suspected := suspected\{s};
        byzantinesender := TRUE;
        forall q ∈ correct do
            trigger ⟨ al, Send | q, [BYZANTINESENDER, echos] ⟩;
    elseif suspected ≠ ∅ then
        suspected := suspected\{s};
        forall p ∈ suspected do
            faulty := faulty ∪ {p};
            correct := correct\{p};
            suspected := suspected\{p};
```

```
upon exists m ≠ ⊥ such that #(p ∈ correct|readys[p] = m) > f and sentready = FALSE do
    sentready := TRUE;
    forall q ∈ correct do
        trigger ⟨ al, Send | q, [READY, m] ⟩;
    StartTimer(Δ);

upon event ⟨ al, Deliver | p, [BYZANTINESENDER, echos_p] ⟩ do
    senderechos := senderechos ∪ {echos_p[s]};

upon #(senderechos) = #(correct\suspected) − 1 do
    if byzantinesender = FALSE then
        forall m ∈ senderechos do
            if echos[SELF] ≠ m then
                forall q ∈ suspected\{s} do
                    correct := correct ∪ {q};
                    f := f − 1;
                f := f + 1;
                faulty := faulty ∪ {s};
                correct := correct\{s};
                suspected := suspected\{s};
                byzantinesender := TRUE;
                forall p ∈ correct do
                    trigger ⟨ al, Send | p, [BYZANTINESENDER, echos] ⟩;
                break

upon event Timeout(Δ) do
    forall q ∈ suspected do
        faulty := faulty ∪ {q};
        correct := correct\{q};
        suspected := suspected\{q};
    suspected := suspected\{s};

upon exists m ≠ ⊥ such that #(p ∈ correct|readys[p] = m) > 2f and delivered = FALSE do
    delivered := TRUE;
    trigger ⟨ abrb, Deliver | s, m ⟩;
```

5 Implementation Environment

The Adaptive Authenticated Double-Echo Broadcast algorithm was implemented using the Python language and all above examples were experimented[2]. A process is characterized by a class *Process* and the events are abstracted using appropriate functions that are called sequentially, respecting the order of the algorithm.

```python
# Adaptive Authenticated Double-Echo Broadcast

from collections import Counter
import time

class Process:
    def __init__(self, id, f):
        self.sentecho = False
        self.sentready = False
        self.delivered = False
        self.echos = {}
        self.readys = {}
        self.id = id
        self.correct = []
        self.suspected = []
        self.faulty = []
        self.f = f
        self.s = ""
        self.byzantinesender = False
        self.senderechos = []
        self.timer = 0
        self.phase1 = False
        self.phase2 = False
        self.phase3 = False
        self.message = ""

    def setProcesses(self, Processes):
        self.Processes = Processes
        self.correct = Processes.copy()

    def setSender(self, sender):
        self.s = sender

    def getId(self):
        return self.id

    def abrbBroadcast(self, msg):
        print("*Broadcast*: " + str(self.id) + " sends " + msg)
        for q in self.correct:
            print("Sending alSend: " + str(self.id) + " -> "
```

```
                + str(q.getId()))
            q.alSendSend(self.id, self, msg) # Trigger alSendSend

    def alSendSend(self, id, sender, msg):
        self.phase1 = True

    def getPhase1(self):
        return self.phase1

    def alDeliverSend(self, id, sender, msg):
        if self.sentecho == False:
            print("Delivering alSend by " + str(self.id))
            self.sentecho = True
            for q in self.correct:
                print("Sending alEcho: " + str(self.id) + " -> " + str(q.getId()))
                q.alSendEcho(self.id, msg) # Trigger alSendEcho

    def alSendEcho(self, id, msg):
        self.phase2 = True
        self.alDeliverEcho(id, msg)

    def getPhase2(self):
        return self.phase2

    def alDeliverEcho(self, id, msg):
        if self.echos.get(id) is None:
            print("Delivering alEcho by " + str(self.id))
            self.echos[id] = msg

    def consistentMessage(self, echos_set):
        msg = {}
        for id in echos_set.keys():
            if echos_set.get(id) not in msg:
                msg[echos_set.get(id)] = 1
            else:
                msg[echos_set.get(id)] = msg[echos_set.get(id)] + 1
        cons, count = Counter(msg).most_common(1)[0]
        return cons

    def suspectedAppend(self, id):
        for p in self.Processes:
            if p.getId() == id:
                self.suspected.append(p)

    def consistentEchoWaitEvent(self):
```

```
        if len(self.echos) > ((3*len(self.correct))/4):
            cons = self.consistentMessage(self.echos)
            print(self.id + " found cons = " + cons)
            for p in self.echos:
                if self.echos[p] != cons and p != self.s.getId() and p not in self.suspected:
                    self.suspectedAppend(p)
                    self.f = self.f + 1
            if self.echos[self.s.getId()] != self.echos[self.getId()]:
                self.f = self.f + 1
                self.faulty.append(self.s)
                self.correct.remove(self.s)
                self.byzantinesender = True
                print(self.getId() + " detected the sender as Byzantine")
                for q in self.correct:
                    q.ByzantineSenderSend(self.getId(), self.echos) # Trigger ByzantineSender
            elif self.suspected is not None and self.s not in self.suspected:
                self.suspected.append(self.s)

    def correct_minus_suspected(self):
        correct_minus_suspected = self.correct.copy()
        if self.suspected is not None:
            for p in self.suspected:
                if p in correct_minus_suspected:
                    correct_minus_suspected.remove(p)
        return correct_minus_suspected

    def readyWaitEvent(self, msg):
        if len(self.echos) > ((len(self.correct) + self.f) / 2) and self.sentready == False and self.byzantinesender == False:
            self.sentready = True
            correct_minus_suspected = self.correct_minus_suspected()
            correct_minus_suspected.append(self.s)
            for q in correct_minus_suspected:
                print("Sending alReady: " + str(self.id) + " -> " + str(q.getId()))
                q.alSendReady(self.id, msg) # Trigger alSendReady
            return True
        return False

    def alSendReady(self, id, msg):
        self.phase3 = True
```

```python
        self.alDeliverReady(id, msg)

    def getPhase3(self):
        return self.phase3

    def faultyAppend(self, collection, id):
        for p in collection:
            if p.getId() == id:
                self.faulty.append(p)

    def correctRemove(self, id):
        for p in self.correct:
            if p.getId() == id:
                self.correct.remove(p)

    def suspectedRemove(self, id):
        for p in self.suspected:
            if p.getId() == id:
                self.suspected.remove(p)

    def alDeliverReady(self, id, msg):
        if self.readys.get(id) is None and self.echos.get(id) == msg:
            print("Delivering alReady by " + str(self.id))
            self.readys[id] = msg
        elif self.readys.get(id) != self.echos.get(id) and id != self.s.getId():
            self.f = self.f + 1
            self.faultyAppend(self.correct, id)
            self.correctRemove(id)
            self.suspectedRemove(id)
        if len(self.echos) > self.f and self.sentready == False:
            self.sentready = True
            for q in self.correct:
                q.alSendReady(self.id, msg) # Trigger alSendReady
            self.timer = time.time() # StartTimer()

    def readyCheckWaitEvent(self):
        if len(self.readys) > 2*self.f and self.byzantinesender == False:
            if self.readys.get(self.s) != self.echos.get(self.s):
                self.f = self.f + 1
                self.faulty.append(self.s)
                self.correct.remove(self.s)
                self.byzantinesender = True
                print(self.getId() + " detected the sender as
```

```
                    Byzantine")
                for q in self.correct:
                    q.ByzantineSenderSend(self.getId(), self.echos) # Trigger ByzantineSender
            elif self.suspected is not None:
                self.suspected.remove(self.s)
                for p in self.suspected:
                    self.faulty.append(p)
                    self.correct.remove(p)
                    self.suspected.remove(p)

    def ByzantineSenderSend(self, p_id, echos_p):
        self.ByzantineSenderDeliver(p_id, echos_p)

    def ByzantineSenderDeliver(self, p_id, echos_p):
        self.senderechos.append(echos_p[p_id])

    def suspected_minus_sender(self):
        suspected_minus_sender = self.suspected.copy()
        if self.s in suspected_minus_sender:
            suspected_minus_sender.remove(self.s)
        return suspected_minus_sender

    def misledCheckWaitEvent(self):
        correct_minus_suspected = self.correct_minus_suspected()
        if len(self.senderechos) == len(correct_minus_suspected) - 1:
            if self.byzantinesender == False:
                for m in self.senderechos:
                    if self.echos.get(self.getId()) != m:
                        suspected_minus_sender = self.suspected_minus_sender()
                        for q in suspected_minus_sender:
                            self.correct.append(q)
                            self.f = self.f - 1
                        self.f = self.f + 1
                        self.faulty.append(self.s)
                        self.correct.remove(self.s)
                        self.suspected.remove(self.s)
                        self.byzantinesender = True
                        print(self.getId() + " detected the sender as Byzantine")
                        for p in self.correct:
                            p.ByzantineSenderSend(self.getId(), self.echos) # Trigger ByzantineSender
                        break

```

```
    def Timeout(self):
        for q in self.suspected:
            self.faulty.append(q)
            self.correct.remove(q)
            self.suspected.remove(q)
        self.suspected.remove(self.s)

    def abrbDeliver(self, msg):
        if len(self.readys) > 2*self.f and self.delivered ==
           False:
            self.delivered = True
            print("*Delivering Broadcast*: " + str(self.id) +
                " recieves " + msg)
            self.message = msg # Trigger abrbDeliver

    def getMessage(self):
        return self.message
```

Listing 1: Adaptive Authenticated Double-Echo Broadcast Algorithm

6 Conclusions

The algorithm discussed adapts its behavior depending on the number of Byzantine processes detected into the system. Although, even if there is a Byzantine, it cannot be detected until it deviates the correct execution of the algorithm. Furthermore, an adaptive protocol cannot be implemented in an asynchronous environment, because the detection of a process as faulty can happen even if the process is correct. For example, it may happen that a process is detected as faulty, while actually it is busy and it can respond to a request only after a certain amount of time, which is finite but also unpredictable.

References

[1] Gabriel Bracha. “Asynchronous Byzantine Agreement Protocol”. In: *Information and Computation* 75 (Sept. 1987), pp. 130–143. DOI: https://doi.org/10.1016/0890-5401(87)90054-X.

[2] Giuseppe Daidone. *Adaptive Byzantine Reliable Broadcast*. URL: https://github.com/GiuseppeDaidone/AdaptiveByzantineReliableBroadcast.