Asynchronous Byzantine Agreement Protocols

Lorenzo Masciullo [203315]

Valentina Odorizzi [203314]

January 12, 2020

1 Introduction

The agreement among processes is important in the distributed systems for several applications. Indeed, sometimes is required that the processes must exchange information in order to reach eventually a common agreement. An example of an application is the commit decision in a database system where the processes decide whether to commit or abort a transaction.

In our case, we implement a model of a distributed system where the processes exchange messages on a reliable channel in order to communicate that is based on [Gabriel Bracha, 1987]. The exchanged messages are not lost or generated. Moreover, every process can send a message to any other process. The processes can be *correct* or *faulty.* A correct process performs correct actions, instead a faulty process may deviate from the correct behaviour. Therefore, we need to a protocol that is *t-resilient*, hence it satisfies the following properties in the presence of up to t faulty processes:

- **Agreement**: all *correct* processes decide on the same value.
- **Validity**: if the all *correct* processes start with the same value v , then all correct processes decide on v.
- **Termination**: Each *correct* process must eventually decide on a value.

The faulty processes can be:

- **Fail-Stop**: the processes could omit messages at any time, then they die and finally stop to participate in the protocol.
- **Byzantine**: the process could deviate from the normal behavior of the protocol. In detail, they can omit or send contradictory messages.

The proposed model is composed by two primitives: *Reliable Broadcast* and *Validate.*

The Reliable Broadcast primitive allows to send the same proposed value to each node. Moreover, it forces the faulty processes sending the same message to all correct processes.

Instead, the Validate Primitive implements a validation method that forces the faulty processes to send only the messages that have been sent by correct processes.

The implementation is described in details in the Section 2.

2 Algorithm implementation

In this section, the implementation of the algorithm will be described. Moreover, it will present the formal model.

The protocol involves two primitives which are *Reliable Broadcast* and *Validate* as introduced in the Section 1.

2.1 Formal model

2.1.1 Process

A process p is a process which contains the following variables:

- v_p: the value proposed by p.
- r_p: the round in which p resides.
- d_p: the final decision of p.

2.1.2 Message

A process p broadcasts a message m to all process *all* in the network. The message's format is represented by the following tuple:

$$\mathrm{m}_{p \to all} = (p,\ type,\ v,\ r,\ l)$$

- p is the original sender p.
- *type* defines the type of message, such as *INITIAL*, *ECHO*, *READY*.
- v is the **value** proposed by p.
- r is the **round** where p sends the message.
- l is a tagged value to indicate that the process p is ready to decide *value* in a particular phase.

2.2 Reliable Broadcast Protocol

The *Reliable Broadcast* Protocol is useful in the considered model since it restricts the behaviour of the Byzantine processes forcing them to send the same message to all processes. In this way, it is considered as a primitive and each of its instances has to consider an independent execution flow. The primitive is respected if and only if it supports the following properties:

- if p is correct, then all correct processes agree on the value of its message;
- if p is faulty, then either all correct processes agree on the same value or none of them accepts any value from p.

The *Broadcast Primitive* defines three different types of message that are used in the protocol: *initial*, *echo* and *ready*.

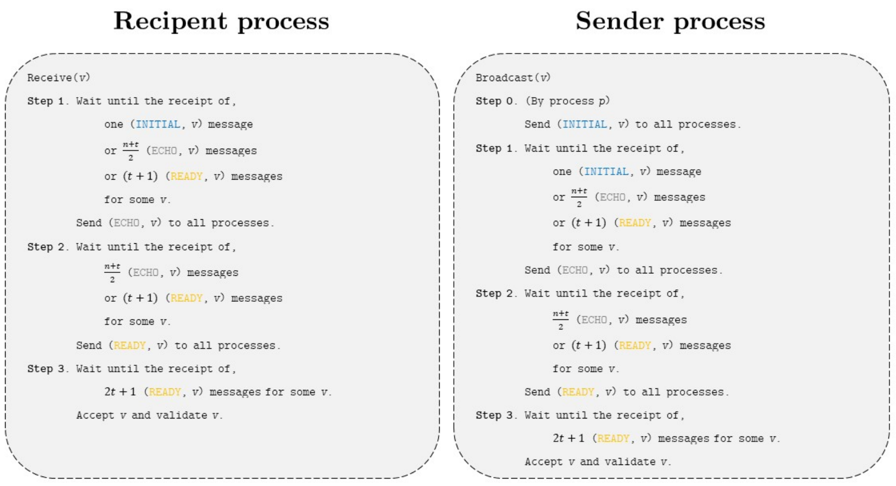
Initially, when a process p starts a new instance of broadcast, it becomes directly the originator sender. Thus, the p broadcasts an *initial* message containing the proposed value v.

The recipient receives the initial message *(initial, v)* and sends an *echo* message *(echo, v)* in order to declare which it knows that p proposes v. An *echo* message can be also sent when the process receives enough echo (i.e. $(n+t)/2$) or ready (i.e. $t+1$) messages that confirm the value v. We consider n the number of all processes and t the maximum number of acceptable faulty processes.

Finally, a process sends a *ready* message *(ready, v)* to declare that p sent only the value v. The *ready* message can be also sent when the recipient receives enough echo (i.e. $(n+t)/2$) or ready (i.e. $t+1$) messages.

A value v is in **accept** state when a process receives enough ready messages (i.e. $2 \times t+1$).

Therefore, the Broadcast primitive is divided in several steps based on the types of message. The Figure 1 shows the steps described in in this section. We have defined the steps for both the sender and the recipient. Since, if we consider a single instance of broadcast, hence one single process broadcasts a proposed value, then the recipients start from step 1. Indeed, a recipient,that receives an initial message, should able to reply with an echo message skipping step 0 and re-starting from step 2.



Recipent process

```
Receive($v$)
Step 1. Wait until the receipt of,
            one (INITIAL, $v$) message
            or $\frac{n+t}{2}$ (ECHO, $v$) messages
            or $(t+1)$ (READY, $v$) messages
            for some $v$.
        Send (ECHO, $v$) to all processes.
Step 2. Wait until the receipt of,
            $\frac{n+t}{2}$ (ECHO, $v$) messages
            or $(t+1)$ (READY, $v$) messages
            for some $v$.
        Send (READY, $v$) to all processes.
Step 3. Wait until the receipt of,
            $2t+1$ (READY, $v$) messages for some $v$.
        Accept $v$ and validate $v$.
```

Sender process

```
Broadcast($v$)
Step 0. (By process $p$)
        Send (INITIAL, $v$) to all processes.
Step 1. Wait until the receipt of,
            one (INITIAL, $v$) message
            or $\frac{n+t}{2}$ (ECHO, $v$) messages
            or $(t+1)$ (READY, $v$) messages
            for some $v$.
        Send (ECHO, $v$) to all processes.
Step 2. Wait until the receipt of,
            $\frac{n+t}{2}$ (ECHO, $v$) messages
            or $(t+1)$ (READY, $v$) messages
            for some $v$.
        Send (READY, $v$) to all processes.
Step 3. Wait until the receipt of,
            $2t+1$ (READY, $v$) messages for some $v$.
        Accept $v$ and validate $v$.
```

Figure (1) Steps defined in the Broadcast Primitive

2.3 Validate Primitive

The validate primitive is useful in order to control the content of the messages sent by Byzantine processes. In fact, this primitive forces the Byzantine processes to send the same message that the other processes already have in agreement.

A process chooses a new value according to the number of round r and a set denoted by $VALID_p^r$. In details, $VALID_p^r$ is a set of messages, called *validate set*, which contains the messages that are accepted at the round r of a specific process p. Therefore, the $VALID$ set is formally defined as follows:

$$VALID_p^r = \{m \textit{ where } m \textit{ is } (q,\ type,\ v,\ r,\ l) \textit{ s.t. } m \textit{ is accepted and } v \in \{0{,}1\}\}$$

The process p can collect the message *(q, type, v, r, l)* in $VALID_p^r$ if and only if (q, type, v, r, l) is accepted and there exist *n-t* messages in $VALID_p^{r-1}$. The processes update their $VALID$ sets whether they accept a message. A message m is validated by process p if $m \in VALID_p^r$; messages that are not in the set are ignored. Therefore, a message m is validated only if it is sent by a correct process at round r.

2.4 Consensus Protocol

Our implementation is based on the *Consensus Protocol* described in the original paper, which uses the *Reliable Broadcast* and *Validate* primitives in order to implement a n/3-resilient consensus protocol.

The consensus protocol is composed by phases which are executed by all processes. A phase is formed by three rounds that can be repeated more than once until the processes decide a value v.

We denote using i the number of phase.

In the round *3i + 1*, firstly the process p broadcasts the message m. Secondly, if the process p has validated *n-t* messages of the round *3i + 1*, then it sets its v_p equal to the majority value of *n-t validated* messages.

In the round *3i + 2*, firstly the process p broadcasts the message m. Secondly, if the process p has validated *n-t* messages of the round *3i + 2*, then it checks the number of messages that have the same value v in order to set its value v_p.

$$v_p = \begin{cases} \textit{(v,d) If more than n/2 messages have the same value v} \\ v_p \textit{ Otherwise} \end{cases}$$

In detail, the tagged value *(v,d)* is used to indicate that the process p is ready to decide. In our case, the tagged value is divided in two parts, the value v and the label l. The label l is always defined in the message m for convenience. Therefore, we use **false** as value of l to define the normal value v and **true** to define the tagged value. Moreover, when the label l is set **true**, it indicates that the proposed value is confirmed by the majority of processes.

In the round *3i + 3*, the firstly the process p broadcasts the message m. Secondly, if the process p has validated *n-t* messages of the round *3i + 3*, then it checks whether *2t* validated messages contains true as value of l in order to decide the value v, so $d_p = v$. However, if there are less messages with the tagged value, then it sets its value v_p.

$$v_p = \begin{cases} v \textit{ If more than t validated messages have true as label } l \\ \textit{random value between 0 and 1 Otherwise} \end{cases}$$

When a decision is made the protocol terminates.

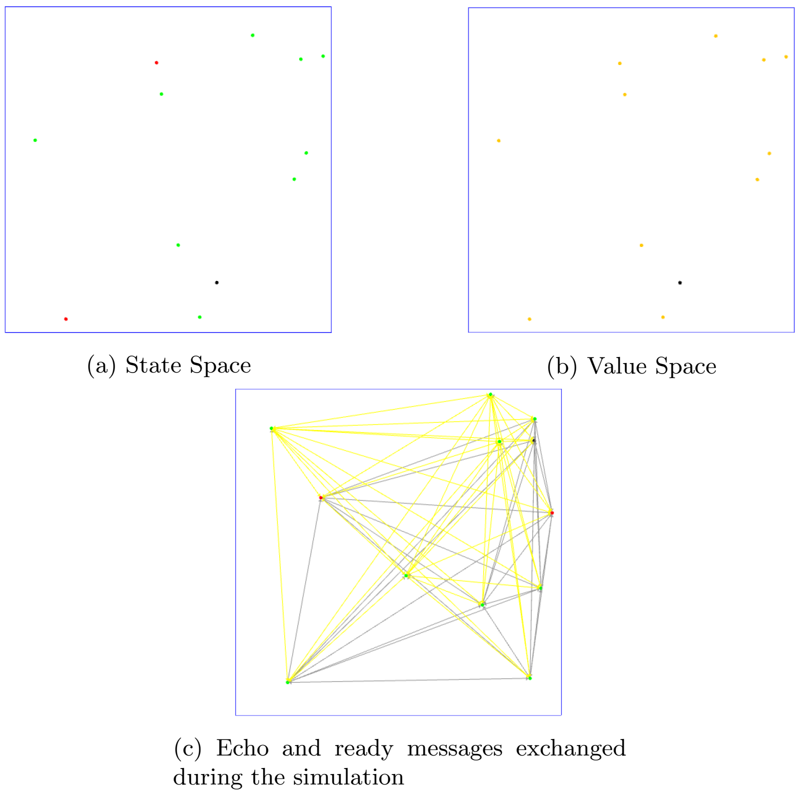
In all round, the process should wait *n-t* messages before perform some actions, this is a particular constraint which could extend the termination times in the presence of Byzantine processes.

2.5 Implementation

In our implementation [1] we have considered three main Agents defined by the following classes: *Process*, *FailAndStop* and *ByzantineProcess*.

The *Process* class is used to define the correct processes. Instead, the *FailAndStop* class is used to represent the Fail-Stop processes. Finally, the *ByzantineProcess* class is used to define the Byzantine processes. Moreover, we use the heredity, in fact Fail-Stop and ByzantineProcess classes extends Process class.

The simulation is composed by two different *continuousSpace* Objects: state space and value space. The state space shows the state of each process which is represented by different colours: green is the correct process, black is the Fail-Stop process and red is the Byzantine process. Instead, the value space shows the value proposed and decided by each process as a color. Indeed, if the process is orange, then it decides 1. Otherwise, if it is blue, then it decides 0. Moreover, in both spaces are displayed the messages exchanged which are edges that are differentiated by colors according to the type: blue (initial), gray (echo) and yellow (ready). The spaces and the messages exchanged during the simulation are shown in Figure 2.


(a) State Space

(b) Value Space

(c) Echo and ready messages exchanged during the simulation

Figure (2) The continuousSpace

Initially, all processes choose a random value between 0 and 1 and broadcast it as proposed value sending a initial message. Each broadcast performed by a process is considered single instance.

For each process, several information is stored in different hash-map, such as the steps (described in Section 2.2), the counters and the winner. The steps and the counters are identified by a couple <*process id*, *round*>. The counters hash-map is used in order to retrieve the number of echo and ready messages sent by the other process related to a particular broadcast instance, so it is used in order to progress with the steps described in the Section 2.2.

The important parts of our implementation are *update validate set* and *check validate set* functions. The *update validate set* implements the validate primitive, so it adds the accepted messages in the valid set, which is an hash-map where the key is the number of round r. In the *update validate set*, the validate set is updated considering the proposed value that is suggested by the majority of processes at the round r-1, so the winner of the previous round.

Instead, the *check validate set* implements the checks described in the Section 2.4. First of all, the *check validate set* retrieves the validated messages, if they are n-t, then the process sets the value based on the round r. In detail, in the round $3i + 1$ of 2.4, the 0 is chosen whether the numbers of proposed values 0 and 1 are equal.

Furthermore, if a decision has not yet been made, then the process p would saves the winner value v in the winner hash-map and progress to the round r+1.

3 Analysis

The main important goal is satisfying the three properties of the protocol, that are *Agreement*, *Validity* and *Termination*. Moreover, we analyse the performance of our protocol in terms of phases.

We have gathered the data from the simulation and, then we have elaborate the information in order to obtain the graphs.

3.1 Parameters

The evaluation of our analysis is based on several parameters which allow to demonstrate the properties and the performance of our protocol. The parameters are the following:

- *Number of Fail-Stop processes*: total number of Fail-Stop processes.
- *Number of Byzantine processes*: total number of Byzantine processes.
- *Number of processes*: number of faulty and correct processes.

  The number of faulty processes is the sum of Fail-Stop and Byzantine processes.
- *Workload per node (Max)*: the maximum number of operations that a process can perform in a single step.
- *Workload per node (Min)*: the minimum number of operations that a process can perform in a single step.

  The workload decreases the efficiency of the protocol with low values, since the broadcast is spread slowly. Anyway, it will be decided in a random way respecting the declared bounds

3.2 Charts

Validity and Agreement

> *Validity*: if the all correct processes start with the same value v, then all correct processes decide on v.

We consider that the validity property is also satisfied replacing *all correct processes* with *majority of correct processes*, since before deciding our protocol examines the values and stores the value that is most voted.

First of all, we have analysed the network considering only the correct processes, so the number of faulty processes t has been set 0.

The Table 1 indicates the value of parameters used during the simulation.

We have highlighted four main elements which are *0-False*, *1-False*, *0-True* and *1-True*. The boolean value represents the label which is sent together with the proposed value. The label indicates that the

Number of processes	10	30	50
Parameter Name			
#Fail-Stop processes	0	0	0
#Byzantine processes	0	0	0
#processes	10	30	50
Workload	100	100	100

Table (1) Parameters' value used in the evaluation of Validity and Agreement properties

proposed value is confirmed by the majority of processes, hence they are ready to accept that value at the next round as described in the Section 2.4.

Figures 3, 4 and 5 show that the processes decide a particular value after three rounds. The value is decided when all correct processes vote the same value v and in the third round, the value is sent with a label set at *True*. In particular, we can notice that in the Figures 3 and 5, the decided value is chosen by the majority. Moreover, if half of the correct processes propose 0 as value and another half of correct processes propose 1, then the value that is propagated among all processes is 0 from our decision, as it is shown in Figure 4.

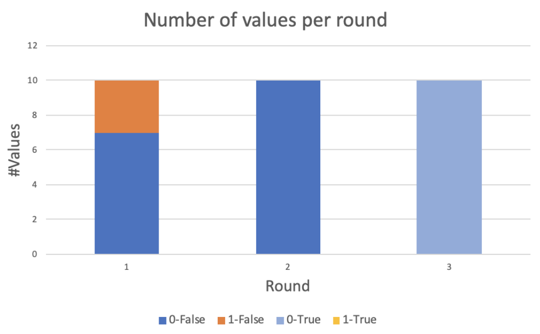


Figure (3) The number of values per round with 10 correct processes

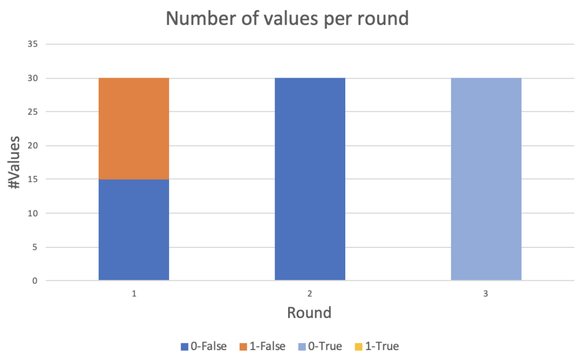


Figure (4) The number of values per round with 30 correct processes

We have also evaluated the network considering the number of faulty processes t equal to the maximum acceptable bound, that is *(n/3) - 1*. First of all, we consider the Fail-Stop as unique faulty processes. Table 2 shows the values used during the simulation. We can notice that in Figures 6, 7 and 8 some proposed values miss due to Fail-Stop processes, which do not send any messages and then crash. In spite of the presence of faulty processes, the protocol is able to decide a final value.

Then, we have performed the analysis considering only the Byzantine as unique faulty processes. However, we had to consider less Byzantine processes than the maximum acceptable bound, since the simulation takes more than 150 phases before terminating due to the protocol's constraints. This is a disadvantage to our protocol.

Table 3 indicates the values of parameters used during the simulation.

In Figure 10, the decision is made by correct processes after 24 rounds, hence 8 phases.

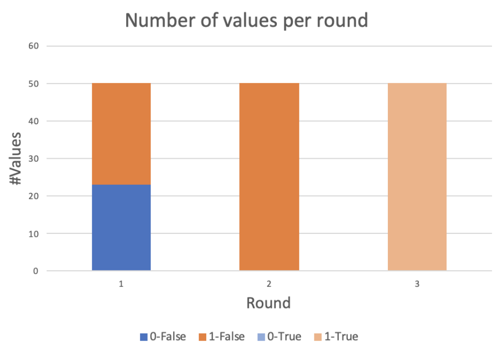


Figure (5) The number of values per round with 50 correct processes

Number of processes	10	30	50
Parameter Name			
#Fail-Stop processes	2	9	15
#Byzantine processes	0	0	0
#processes	10	30	50
Workload	100	100	500

Table (2) Parameters' value used in the evaluation of Validity and Agreement properties with Fail-Stop processes

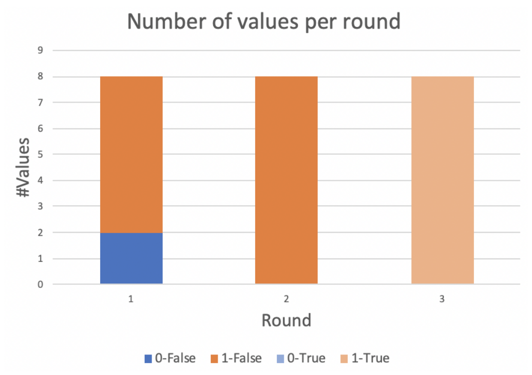


Figure (6) The number of values per round with 8 correct processes

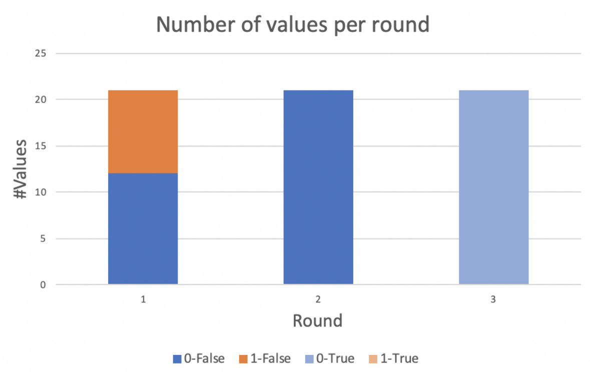


Figure (7) The number of values per round with 21 correct processes

There is a pattern that is constantly repeated after the third round due to the valid set. In fact, the processes do not decide until the validate set contains enough labeled values (in this case 18, which is $2t$). However, we can notice that the Byzantine processes do not affect the decision of the correct processes.

Also in Figure 11, the correct processes take several phases before deciding. However, the Byzantine processes do not affect the decision of correct processes. Indeed, the correct processes always propose

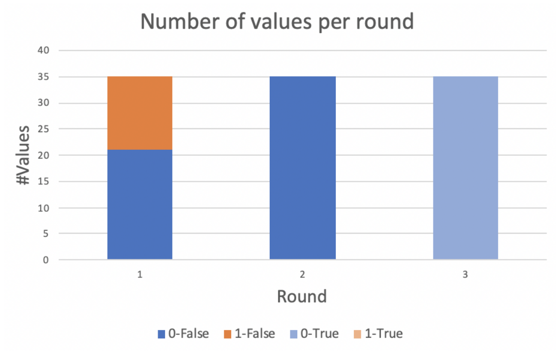


Figure (8) The number of values per round with 35 correct processes

Number of processes	10	30	50
Parameter Name			
#Fail-Stop processes	0	0	0
#Byzantine processes	2	7	10
#processes	10	30	50
Workload	100	100	500

Table (3) Parameters' value used in the evaluation of Validity and Agreement properties with Byzantine processes

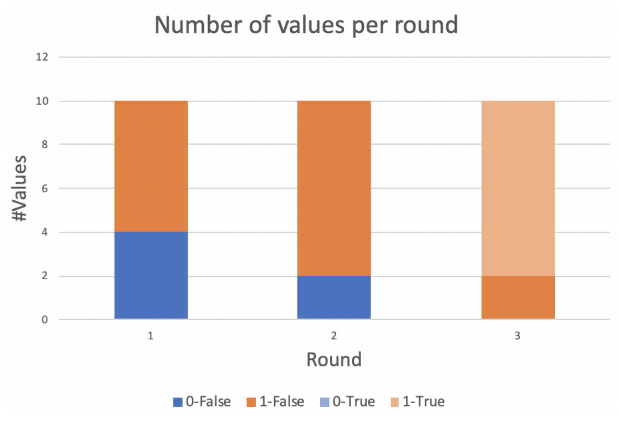


Figure (9) The number of values per round with 8 correct processes

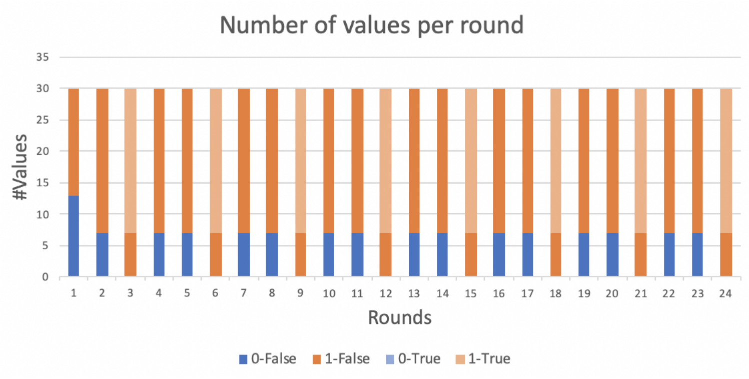


Figure (10) The number of values per round with 23 correct processes

1 as value (which is the most proposed value at round 1) even if the byzantine processes propose 0 as value.

The case with t is greater than *(n/3) - 1* is described in the Termination Analysis Section.

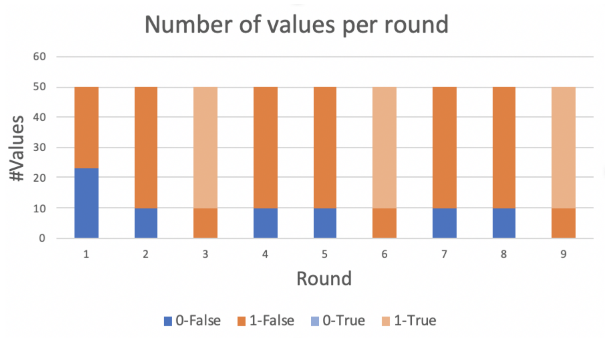


Figure (11) The number of values per round with 40 correct processes

Termination
The evaluation of termination property is based on the data set gathered at the previous analysis. Therefore, the charts in this Subsection are related to the previous graphs.

First of all, we consider the graphs related to the parameters in the Table 1.

We can notice in Figures 12, 13 and 14 that the protocol terminates for all the three simulations, so all processes decide a value v. In detail, they decide the value that is the most proposed.

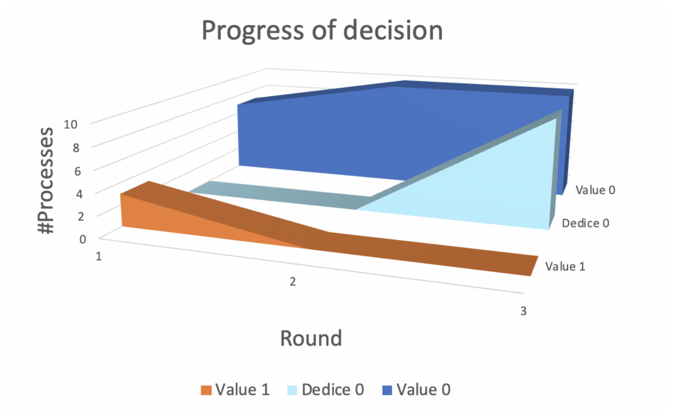


Figure (12) 10 correct processes decide the value 0

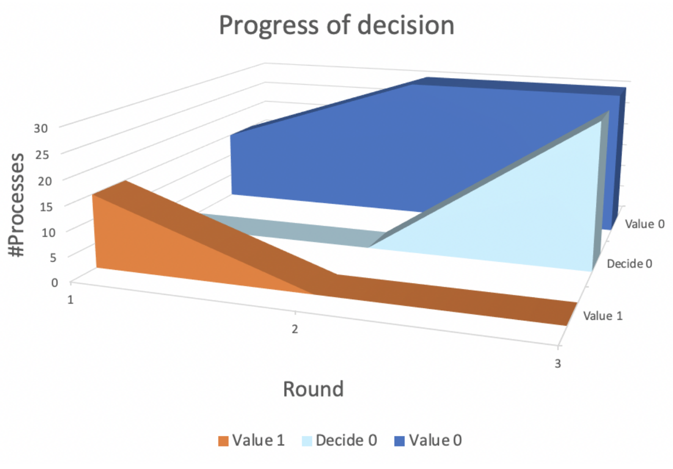


Figure (13) 30 correct processes decide the value 0

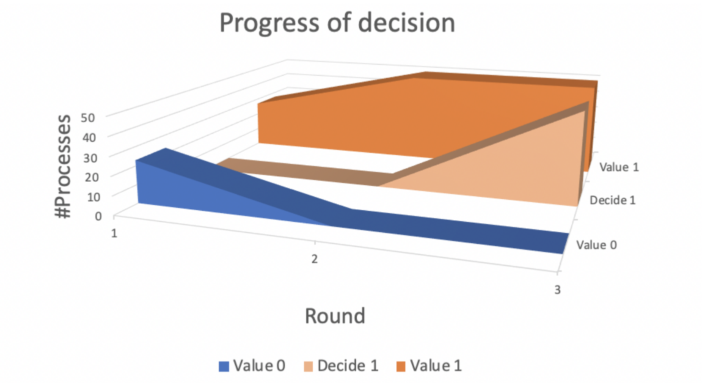


Figure (14) 50 correct processes decide the value 1

We also confirm the termination property in presence of Fail-Stop processes. The Fail-Stop processes do not decide any value, but the termination property is satisfied by the correct processes.

Figures 15, 16 and 17 show the progress of decision when the number of faulty processes is set at the maximum acceptable bound.

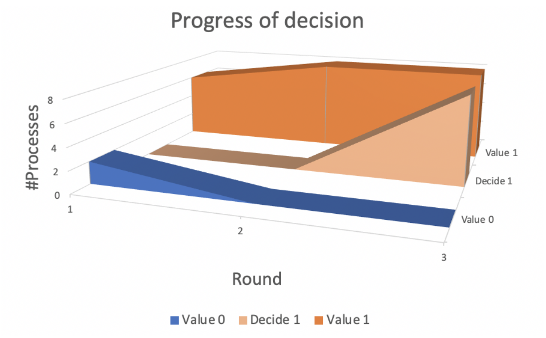


Figure (15) 8 correct processes decide the value 1

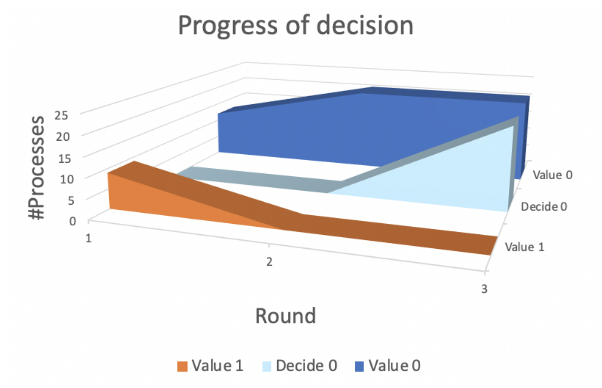


Figure (16) 21 correct processes decide the value 0

A decision is also made when the simulation contains several numbers of Byzantine processes, which may modify the final decision. Initially, they are considered correct processes since all processes propose random values.

Therefore, we can notice in Figures 18, 19 and 20 that all processes eventually decide. Moreover, the decided value is coherent with the progress of the proposed value during the simulation. Hence, the protocol is able to maintain the integrity of the agreement.

In detail, in Figures 19 and 20, the proposed value 0 trace a wave line due to the behaviour of Byzantine processes. Instead, the proposed value 1 is constant after the second 2 since we only consider the value with the label set at true. However, both correct and Byzantine processes decide at the end.

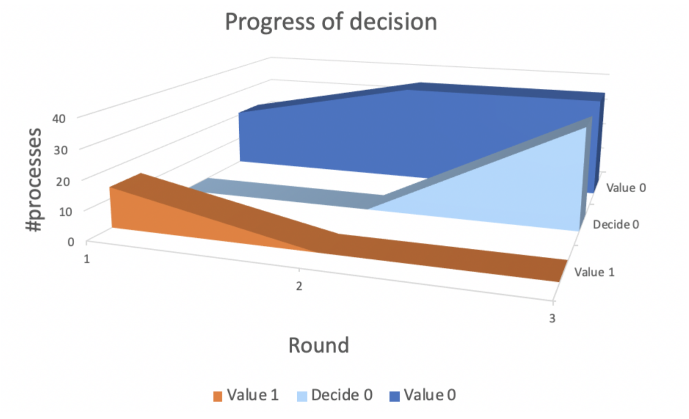


Figure (17) 35 correct processes decide the value 0

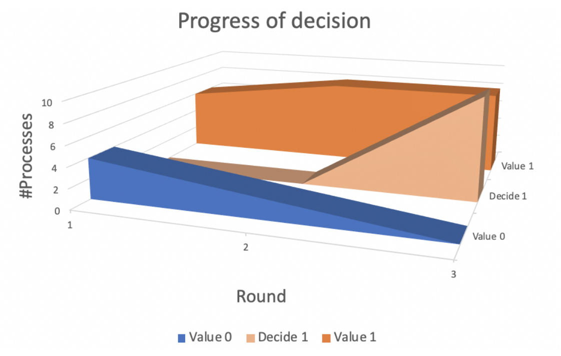


Figure (18) 8 correct processes decide the correct value even the presence of Byzantine processes

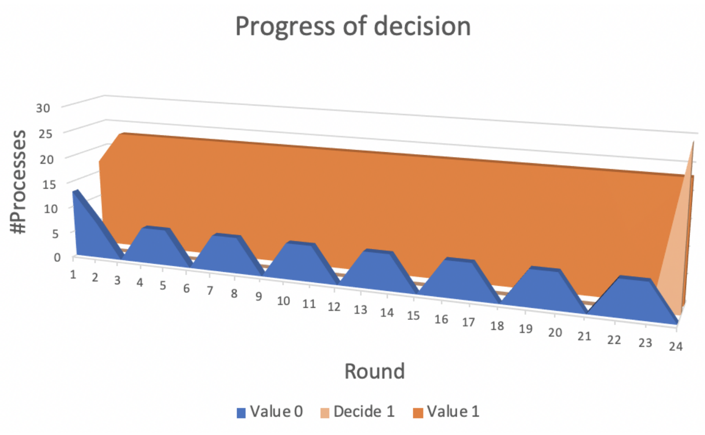


Figure (19) 21 correct processes decide the correct value even the presence of Byzantine processes

The simulation with 30 processes has more phases than the simulation with 50 processes since the number of Byzantine processes is not proportionate to the total number of processes. Therefore, the initial configuration is not correctly balanced. Indeed, the Byzantine processes affect the validate set of the simulation with 30 processes with more force.

Finally, if we consider the case with the number of faulty processes greater than the maximum acceptable limit (i.e. *(n/3) - 1*), then the protocol can have unstable behaviour. For example, if we exceed the bound with the number of Fail-Stop processes, then our protocol waits until the validate set contains *n-t* messages, so endlessly. In the case of Byzantine processes, the protocol can work correctly or some correct processes may vote a value that is proposed by a faulty process.

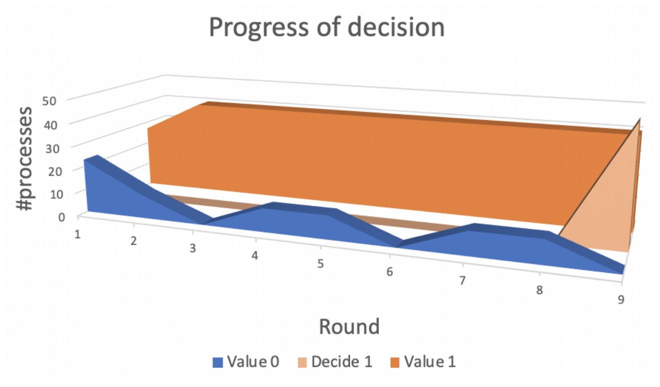


Figure (20) 35 correct processes decide the correct value even the presence of Byzantine processes

Performance

The protocol has a probabilistic behaviour, so it can terminate at the first or at the 50^{th} phase. However, we have noticed that the number of processes affects the termination. Indeed, a simulation with 10 processes takes fewer phases than a simulation with 50 processes. We have analysed this behaviour simulating our protocol considering from 5 to 50 processes. Moreover, we have simulated the network also considering the byzantine processes, which the number is chosen as following described:

$$number\ of\ byzantine\ processes = \lceil (\frac{\frac{n}{3}-1}{2}) \rceil$$

Therefore, we consider a balanced number of byzantine processes for each simulation without stressing the network. Moreover, we have balanced also the workload for each simulation in order to simulate correctly the network. In fact, if we assign 10 as the workload for a simulation with 50 processes, then the processes could not decide easily. Otherwise, if we assign 100.000 as workload, then the processes decide at the first phase.

Figure 21 shows the average of phases per number of processes. For each simulation with n processes, we have considered the average of phases obtained in several simulations in order to not focus on a particular value.

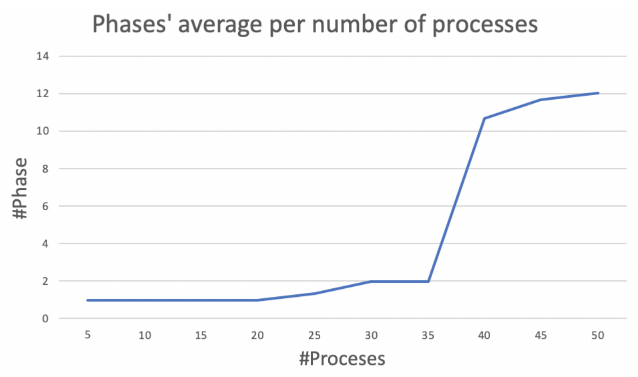


Figure (21) Average of phases per number of processes

As a result, we can notice that the number of phases is related to the number of processes.

4 How to run the simulator

Once download the `setupBConsensus.jar` from the link placed in the repository's `README`, it's possible to follow the wizard steps in order to install all the resources needed to execute the model simulator. Basically, you can execute the following steps in order to run the simulator correctly.

- Make sure *Java RE 1.8* is installed in your system.

- Run `setupBConsensus.jar` using system GUI or typing `java -jar setupBConsensus.jar` on the terminal.
- Follow the wizard steps in order to accept licence and choose the installation directory, as you prefers.
- Launch the simulator executing the `start_model.bat`. A Java application will showed on your screen. Warning: if you use an Apple MAC then you have to run `start_model.command`.
- Set each parameter from the `Parameters` window. If you don't see it, you can reach it clicking on *Window* menu and take it from the *Show View*. Once you have chosen parameters, it's possible to save them in order to not rewrite them on the follow time.
- Finally, you can launch the simulation doing click on the "Initialize Run" button placed on the upper bar. At this point, you can see spaces and graphs in the main window of the simulator. You can start simulation doing click on "Start Run" button and observe results. If you want, it's possible perform simulation slower from "Run Options" window.

5 Conclusion

In conclusion, our implementation satisfies the three main properties of the consensus protocol, such as validity, agreement, and termination. Moreover, it is *n/3-resilient*, so the protocol works correctly when the number of faulty processes is less than *n/3*.

However, it has some disadvantages since the protocol considers the first *n-t* validated messages. The messages could be sent by faulty processes that affect the termination time of the protocol. Hence, the protocol can take several phases before making a decision. Therefore, giving more importance to the first validated *n-t* messages in terms of arrival time, it could not be a good solution.

Moreover, when the number of faulty processes is greater than *n/3*, the protocol does not understand that it is in a deadlock state, since the validate set does not never reach the required size, so it never ends.

References

[Gabriel Bracha, 1987] G.Bracha, T.Gordini, (1987) *Asynchronous Byzantine Agreement Protocols*

[1] V.Odorizzi, L.Masciullo, (A.Y. 2019-2020) *Asynchronous Byzantine Agreement Protocols Implementation* `https://github.com/lore-masc/asynch-byzantine.git`