Sorbonne Universités, UPMC Univ. Paris 06
Master Informatique – Systèmes et Applications Répartis

Efficient handle of byzantine faults in clouds

Bournat Marjorie

Supervisor : Olivier Marin

Referent teacher : Julien Sopena

I. Introduction

Desktop Grid are more and more used to compute tasks or to stock data. *Desktop Grid* is a distributed environment where there are many types of faults. Those faults can be crashes or complex faults like byzantine faults.

Byzantine faults corresponds to arbitrary behavior of machines. They can be due to software mistakes, hardware mistakes or because of attacks. These class of fault is the larger type of faults: it includes crashes and transient faults. The name “byzantine faults” comes from the analogy used by Lamport et al. [1] when he defines them.

Byzantine faults can break a system, and these kind of faults are more and more spread because of the complexity of applications (and so because of the potential bugs permitting intrusions). So techniques have been developed to tolerate them. Two techniques stand. On one hand there is techniques that use hardware to deal with byzantine faults like [2] and on the hand there are techniques using software to tolerate them. The software solutions are divided in two families. The first software techniques use replication. It means that we have got a client asking to a scheduler for the execution of a query, the scheduler doesn't just ask to a worker (machine that have to execute the query/task) to execute this query but asks to several workers to execute it. In fact, if the scheduler asks to execute the query to only one worker, if this worker is byzantine, the scheduler give send probably (it isn't because a worker is byzantine that it will answer badly: the node can try to fool the system giving during a time good response) an incorrect answer to the client. After the execution of the request, the workers give the result to the scheduler which chose the result to send to the client. Very often the result send to the client corresponds to the majority result received from the workers. In the second software method the scheduler forward the query of the client to only one worker. However, it sends to this worker *n* others queries. The scheduler knows the answers of those *n* others queries. When the scheduler receives all the answers (there are *n+1*), it can decides if whether or not the result of the query of the client is correct. In fact, if the result to the *n* requests given by the worker correspond to the *n* answers the scheduler know, it assumes that the query of the client is correct too. It can then send the result to the client. Otherwise it needs to ask the *n+1* queries to an other worker.

We focus on the replication technique.

Algorithms that tolerate byzantine faults (named BFT for Byzantine Fault Tolerant) have multiple goals. They must assure properties of safety and liveness. The safety assures that the system behave beyond its specification and the liveness insures that the execution gets on. Some algorithms besides guarantees the confidentiality of data. We don't care of this last characteristic, because our goal isn't to manipulate critical data but to insures an efficient service to tolerate byzantine faults.

I.1. Principles of the existing BFT

To tolerate byzantine faults thanks to replication there are two ways in the state of the art. There are deterministic methods and probabilistic methods.

deterministic methods:

With deterministic methods (like []) the scheduler always give a correct answer to the client. However to be able to do that, there is an assumption on the system: there are at most *f* byzantine

faults.

So to give a correct result to the client, the scheduler has to replicate its query 3f+1 times. In fact, as there are at most 3f+1 byzantine faults, it is possible that we have requested in the worst case *f* byzantine workers. And as a byzantine fault can be a crash, the scheduler can't wait 3f+1 answers. If the scheduler wait 3f+1 answers and that there is at least one crash, the scheduler will wait infinitely. So the scheduler wait only the answers of 2f+1 workers upon the 3f+1 requested, because f can have crashed. But it is possible that beyond this 2f+1 answers received, *f* responses are wrong. In fact, a non-byzantine worker can be slow, and so it is possible that the 2f+1 answers received corresponds to the answer of *f* byzantine workers and of f+1 non-byzantine workers. Thus, the replication on 3f+1 workers is mandatory to be able to determine the majority answer which is the result given by f+1 of the workers requested.

There are drawbacks in these methods. In fact, replicate on 3f+1 workers is quiet big. Moreover the choice of the f could delicate. In fact, if we chose a value f which is very large compare to the real number of byzantine faults in the system, we will use resource in an unnecessary way. For instance, if we chose f=10, we have to replicate each tasks on 31 machines, and if the real number of byzantine faults is 2, only 7 replications would have been enough to give a correct answer to clients. The 24 others machines could have been used for others tasks, and so the throughput could have been better if f have been sized correctly. On the other way, if we chose a value f smaller than the real number of byzantine faults in the system it is possible to provide false result to the client.

Notes: deterministic methods use 2 phases: one phase of agreement to order the request of the client and a phase of execution to execute the request of client following the order determined during the previous phase.

probabilistic methods:

The probabilistic doesn't assume that there is a maximum number of byzantine faults on the system. However the result returned to the client isn't always correct. Nevertheless it is possible to be quiet sure of the result sent.

To do that the scheduler (which in those algorithms is reliable, it doesn't crash and it isn't byzantine) conserves a reputation of all the workers of the system. A reputation corresponds to an estimation of the reliability of the answers returned by a worker. At first when the scheduler doesn't know a worker it gives it a reputation of 50 over 100. Then during the execution this reputation will evolve: if the worker gives correct answer (a correct answer corresponds for instance to an answer returned by the majority of the workers requested) the scheduler will increase the reputation of the worker, and if the worker gives bad answer the primary will decrease its reputation.

In this way the scheduler can form group considering the reputation of workers. In this way the group form can be sufficiently reliable to assume that the answer send is correct.

This method has however a drawback: there is only one scheduler, and so there is a bottleneck. The scheduler has the receive the request, compute an algorithm to form groups considering the reliability the client want for his request, send the replicated task to the workers and send the final result to the client.

Notes: The probabilistic methods don't consider an agreement phase because they consider that all tasks executed are independent. It means that the system doesn't keep global information of the system that each task can modify (like a NFS will do).

In the deterministic methods there isn't any reliable node. Here we consider a reliable node. The question that we could ask is why don't use this node to execute the tasks of clients, or multiple

reliable nodes to do this work. Reliable nodes are expensive. To have reliable nodes we need for instance to duplicate the hardware, so there is a cost to have a reliable machine. We can't have a lot of reliable nodes, and the tasks we consider are very big computations (these computations can't be done by one node. We consider tasks that we can divide in multiple independent jobs). So we need to request an important number of nodes. To find a big quantity of nodes we request unreliable nodes which are the most expanded type of machines.

I.2. Goals to achieve

First of all we will remember the characteristics we want for our BFT algorithm. We would like to have a BFT algorithm with weak latency, this means that client won't wait a long time before obtaining the answer to there queries. The algorithm has to insure a good throughput. It has to insures elasticity too. The elasticity corresponds to the adaptability of the algorithm in function of the workers present in the system (the nodes can come and leave the system at any time). Each task has to be replicated on the fewest possible nodes. It mustn't make any assumption about the maximum number of faults in the system, and has to insures good performance even if the number of faults increase. Finally the algorithm must be scalable, and so bear a great number of clients keeping good performances.

The table below summarize the state of the art BFT characteristics.

Algorithm	Latency (case without fault)		Number of agreement machines	Number of execution machines	elasticity
	Cost	Response time			
PBFT	$O(f^2)$	5	3f+1	3f+1	No
Quorum de Aliph	O(f)	2	0	3f+1	No
Quorum de HQ	O(f)	2 for readings 4 for writings	0	3f+1	No
[10]	≤ O(f)	2 to 4[1]	1	Depends on the reliability of nodes and of the reliability the client wants for its answer	Yes
Q/U	O(f)	2	0	4f+1 (but needs of 5f+1 machines)	No
Chain de Aliph	O(f)	3f+2	1	3f+1	No
Zyzzyva	O(f)	3	1	3f+1	No
Zlight de AZyzzyva	O(f)	3	1	3f+1	No
UpRight[2]	$O((u+r)^2)$	7	2u+r+1	u+r+1	No

1 In the article the authors don't take into account the client. But if we considere the client we need to additional messages (one message corresponding to the request from the client to the scheduler, and one message corresponding to the answer from the scheduler to the client)

2 In UpRight u corresponds to the number of crashs and r to the number of arbitrary behavior tolerated by the protocol

BFTCloud	O(f)	3	1	3f+1	Oui
ByzID	O(f)	3	1	2f+1	No
[14][3]	$O(f^2 + g*f)$	7 (9 si on utilise un firewall)	3f+1	2g+1	No
[15]	O(g*f)	5 en considérant un seul checkpoint sinon 5+4*nbCheckpoints	3f+1	Entre g+1 et 2g+1	No
[16]	O(f)	3[4]	2f+1	2f+1	No
[17]	$O(f^2)$	5	3f+1	3f+1	Yes
Aardvark	$O(f^2)$	5	3f+1	3f+1	No
Spinning	$O(f^2)$	5	3f+1	3f+1	No

Beyond the BFT of the state of the art no one is scalable. They all have a bottleneck when the number of request increases. In fact all those algorithms rests on only one scheduler (also named *primary*) or on a group of agreements machines charge to wait the requests from clients, to order them on the workers, to wait the answers and to send the result back to the client.

Beyond the state of the art BFT, only the probabilistic one ([10]) don't assume the existence of a limit on the maximum number of byzantine faults. These algorithms have a weak latency and replicate each task on the necessary and sufficient number of workers in order to achieve a certain threshold of reliability on the result wanted by the client. So even if with those methods the result isn't 100% correct, we will focus on the probabilistic techniques.

In order to make the probabilistic BFT algorithm scalable we will develop a distributed architecture. We will compare the state of the art centralized probabilistic BFT algorithms with our hierarchical architecture.

First we will present the system that we consider with the assumptions we do on it. Then we will present the pseudo-code of our solution. After we will describe the algorithm of the state of the art which will serve for the comparison to our solution. Then we will present the simulator we used to implement all the algorithms used. Section VI will deal with the implementation. And finally we will show results of the tests and analyses of those tests.

II. System:

II.1. architecture:

(u ≥ r)

3 In all the table f corresponds to the maximum number of faults beyond the agreement machines and g corresponds to the maximum number of faults beyond the execution machines.

4 To execute the agreement phase there are oracles that broadcast informations at regular interval of times. The response time is 3 if we consider only one broadcast to execute the agreement.

We consider a distributed architecture with multiple reliable schedulers (also named further *primaries*) and several unreliable workers. The architecture in the probabilistic algorithm of the state of the art are flat: there is only one reliable scheduler. So when a client want to send a request, he send it to this *primary.* In the same way, when a worker want to join the system, it send its request to this *primary.* It is reasonable to think that participating nodes (clients and workers) can know the identity of the *primary.* For instance this identity can be reliably hit in the code of the nodes of the system.

In the case of our distributed algorithm, similarly to the centralized algorithm, clients and workers could known the identity of all the *primaries* of the system. This would permit us to have a totally decentralized solution, thus to have a scalable solution. However we want to implement an adaptive solution: we would like to adapt the number of *primaries* depending on the needs of the application. The number of *primaries* increase if the number of *workers* increase and vice versa, the number of *primaries* decrease when the number of *workers* diminish. So it isn't possible to reliably hit the identity of *primaries* on the system entities. So we add a reliable entity named *first-primary.* When the application needs additional *primaries* the *first-primary* contact *primaries* to join the system and save their identities. When there is to much *primaries* compare to the number of *workers*, the *first-primary* suppress some *primaries* and so suppress their identities. Clients who want to send a request now contact the *first-primary* which transfer the request to one of the *primaries* chosen in function of the strategy selected (see section III.7). Similarly a worker which want to join the system ask it to the *first-primary.* Our distributed solution is now a hierarchical centralized solution. Even if there is a centralized entity (the *first-primary*), we won't lost in performance and scalability. In fact, this entity is only a serving hatch and so adopting the correct data structure the *first-primary* can chose the appropriate *primary* rapidly. Thus the performance stay good as if the number of clients increase. And as if the *first-primary* becomes a bottleneck it is totally feasible to add several *first-primaries* which know all the *primaries* in the system at any time. In this study we focus on the architecture describes figure 1 with only one *first-primary*, multiple *primaries*, and several *workers*.

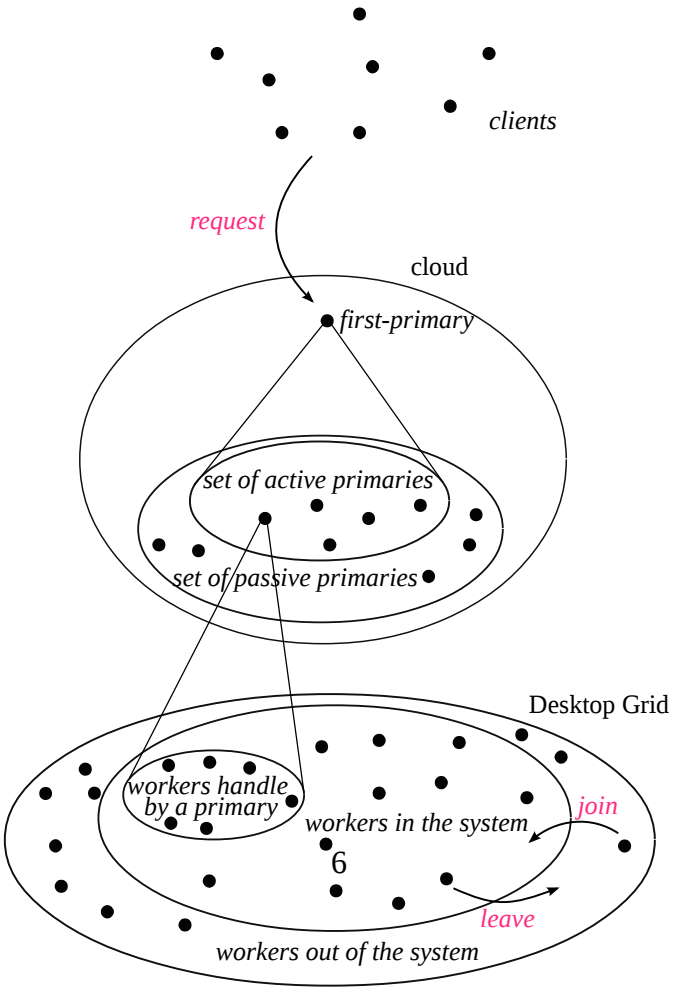

Figure 1 – general architecture of our solution

The *first-primary* and the *primaries* are allocated in the *cloud.* The *cloud* is an administrated environment that permit us to assume in a reasonable way the reliability of the *first-primary* and of the *primaries.*

Our system must treat a big quantity of requests from clients. Those requests correspond to big task that can be divided in multiple smaller independent jobs. We need lots of machines to execute those jobs. The cloud is an environment where resources are payant. Paying to have few reliable entities is reasonable, but this become boring when we want to have an important quantity of machines. So we place our workers in an unreliable environment: *Desktop Grid.* The environment is composed of a great quantity of workers. The nodes of this environment corresponds to machines borrow by people (like you) who have install a program to authorize an extern application to execute itself on their computer. When those people don't use their computer the extern application can use CPU time. But when the owner of the machines need to use his computer the extern application can't execute itself. Some owners authorized the use of their CPU as if they are using it, but the extern application has to execute itself with a low priority, and so the treatment of the application is slower. Thus in function of the use of the compute by the owner the worker can join and leave the system at any time. So the extern has to deal with the dynamism of the workers. Moreover as *Desktop Grid* isn't administrated, the application has to tolerate byzantine faults.

II.2 Assumptions:

First of all we consider that the network is partially synchronous. This assumption is necessary to insure the liveness.

We don't make any assumptions about the maximum number of byzantine faults (crashes included) that occur.

We suppose that all the faults are independent. This means that the faults don't occur because of the same cause. For instance a fault can be due to a software bug while an other can be due to a hardware bug. But two faults can't be caused by the same software bug for example.

We authorize the byzantine workers to enter in collusion, but this isn't mandatory. This means that multiple byzantine workers requested for the same task by the same *primary* can agree to return the same false answer. We have to notice that if all the requested nodes for a task are byzantine, and decide to collude to return an incorrect answer our system is corrupt (it give bad answer to clients). However if the number of byzantine nodes in the system is less than the number of correct nodes, the probability to return bad answer to client because of the collusion of byzantine nodes is weak. Moreover our system is based on a reputation system, which will permit to not request some byzantine nodes (nodes that have give bad answers multiple time will get a bad reputation and so we could stop to request them).

We don't consider the Sybil attack. A Sybil attack is an attack where a node take multiple identity. Thanks to all this identities, the node can be in superiority compare to the others nodes of the system. So giving false answer to the system, this node can corrupt all the system.

III. Our solution's algorithms

We want to write a probabilistic protocol to tolerate byzantine faults. This algorithm must be

scalable. To do that we have decided to have multiple reliable *primaries*. Each *primary* handle a group a *workers* charge to execute the tasks. The intersection between groups handle by *primaries* must be empty. Each *primary* schedule the request of clients on the workers it handles.

So we have to establish the way the workers are distributed on the different *primaries*.

Moreover, our approach is a probabilistic one in order to reduce the number of workers used to replicate a task. A *primary* mustn't interrogate all the workers under its responsibility. So we have to define the different possible strategies to form subset of *workers* beyond the workers handle by a *primary.*

We want that the result return to a client corresponds to a result with a certain reliability defined by the client. So first the *primary* will replicate on the minimal number of nodes necessary to reach the threshold asked. Thus, if all the nodes have answer the same response we could return the result to the client. But if not, we have to replicate the task on additional workers. So we have to introduce the strategies to do some additional replication.

Besides as our workers are in a *Desktop Grid*, workers join (and leave) the system at any moment. We don't know in advance if a *worker* that join the system will be reliable or not. Each *primary* give a reputation to workers its handle in function of their actions all along the execution. So we have to study the strategies permitting to update the reputations. This strategy permits to know what nodes the system can trust.

We consider that nodes can change their behavior. A node that was giving good answer during a long time can suddenly return bad answer. So we will study strategy to avoid this.

Finally we describes the different ways existing for a client to chose the *primary* (via the *first-primary*) he want to request.

All the different strategies will be compared. The goal is to know which one is the best in terms of latency, throughput, and minimal number of *workers* used to replicate a task. We will analyzed these characteristics when there is no byzantine faults in the system and when there is byzantine faults. We will determine which strategy is better when there is faults in terms of result's quality: which strategy permits to keep returning correct results when the number of faults grow up. As the scalability is important for us, we will investigate to know which solution have the best throughput when the number of clients increases.

III.1. Structures used

Now we present the fields we need to write our algorithms. We describe first the fields necessary for the *workers*, then those for the *primaries* and finally those for the *first-primary.*

Each *worker* have a structure W_i :

```
structure W_i {
    W_i.primary     // identity of the primary responsible for the node
}
```

Each *primary* is responsible of a set S of *workers*. So it conserves for each *worker* the information below:

```
structure S_i {
    S_i.a     // answer returned by the node to the previous request
    S_i.reputation      // reputation of the node
    S_i.totR      // number of requests submitted to the node
```

S_i.totC // number of answers given by the node that are assumed correct
}

Each *primary* conserves in memory information about it too. These informations are contained in the structure below:

structure P_i {
P_i.A // set of active *workers* under the responsibility of the *primary*
P_i.P // set of passive *workers* under the responsibility of the *primary*
P_i.tmpSet // set used during reset phases
P_i.minR // minimal reputation of *workers* that the *primary* handles
P_i.maxR // maximal reputation of *workers* that the *primary* handles
P_i.churn // number of the variation of *workers* leaving and joinig the system
P_i.inPool // boolean indicating if the *primary* is having nodes under his responsibility and so receiving requests from client
P_i.NotInF // set of *workers* whose reputation require a change of *primary*
P_i.able_to_send_division // boolean used when executing the *load balancing* function (see notes of section III.2)
P_i.able_to_send_fusion // boolean used when executing the *load balancing* function
}

The intersection between P_i.P et P_i.A is empty all along the execution. When *primaries* are put in the system the value able_to_send_fusion and the value able_to_send_division are put to 1 (indicating that is true, a value of -1 indicate a false value).

The *first-primary* have to keep in memory the *primaries* participating to the application, and the reputations range of the workers they handle. The structure is:

structure FP {
FP.PA // set of active *primaries* with the reputation range of the workers they handle
FP.PI // set of inactive *primaries* with the reputation range of the workers they handle
}

In a real application the field Fp_i.PI hasn't to be present. In fact, when we want to add a node in the active set Fp_i.PA the *fisrt-primary* just has to allocate a new node in the cloud, and when we want to suppress a node from the inactive set Fp_i.PI the *first-primary* just need to suppress it from the system. However we won't deploy our application on a real platform, so we will have a pool of *primaries* put in the Fp_i.PI. Then when the application needs a new *primary* the *first-primary* just take a node from Fp_i.PI and put it in the Fp_i.PA. Similarly when there are too much nodes the *first-primary* will take a node from Fp_i.PA and put it in Fp_i.PI.

III.2. Assignment of *workers* to *primaries* strategies

In this section we describes the different strategies to assign *workers* to *primaries* that we will implement. As *workers* can join and leave at any time we have to take into account these joining and leaving actions.

We propose two assignment strategies: a random strategy and one in function of the reputations of *workers.*

Whatever the strategy used the joining of a node follow the protocol describes figure 2. The *worker* ask to join the system to the *first-primary.* This last compute an algorithm to know towards which *primary* it has to forward the request.

Figure 2 – addition's of a new node protocol

Similarly when a node wants to leave the system whatever the strategy chosen, the node has to inform directly the *primary* responsible for it (unless the node has left the system because of a crash).

III.2.a. Random strategy

The random strategy use a uniform random selection to distribute the *workers* on the *primaries*. The uniform propriety brings two properties on the groups formed: each *primary* has under its responsibility quiet the same number of nodes, and the reputation of each group formed (roughly corresponding to the addition of the reputation of each *workers* belonging to the group) is quiet similar.

The uniform distribution can permit us to make the assumption that when nodes leave the system, it is also uniform. So the two previously introduced properties are always check even when nodes leave. However, we can describe a reset strategy activated after the arrival or the departure of a lots of nodes. The reset insures the two properties stay true. This reset strategy is called *reset* (see the function at the end of this part), and is activated when a *primary* noticed that there are too much nodes that have joined or left compare to the number of nodes it handles initially.

A node can join the system whenever it wants. To do that it has to execute the following algorithm:

---

arrival of a new node

---

init : node W_i wants to join the system
input : identity FP of the first-primary

send (JOIN, W_i) to FP
receive ACKJOIN from P_i
W_i.primary = P_i

upon reception of (JOIN, W_i) from W_i on FP
input : Set P of *primary*

val ← random value in [1, |P|]
send (JOIN, W_i) to P_{val}

upon reception of (JOIN, W_i) from FP on P_i
input : number x of nodes joining or leaving the system autorized before reset

send ACKJOIN to W_i
$P_i.P = P_i.P \cup \{W_i\}$
P_i.churn++
if abs(P_i.churn) > x **then**
call *reset*
end if

Similarly a node leaves the system informing the *primary* responsible for it. Sometimes, some *worker*s crash and thus leave the system without being able to inform its *primary.* To parry that *primaries* have timers which permit the detection of crashes. In this way the *primary* can permits to stop consider nodes which have crashed.

---

upon leaving (or crash detection) of a node

init : node W_i wants to leave the system
send LEAVE to W_i.primary

upon reception of LEAVE from W_i or his crash detection on P_i
input : number x of nodes joining or leaving the system autorized before reset

P_i.churn--
if $W_i \in P_i.P$ **then**
$P_i.P = P_i.P \setminus \{W_i\}$
else
$P_i.A = P_i.A \setminus \{W_i\}$
end if
if abs(P_i.churn) > x **then**
call *reset*
end if

The reset function permits to obtain again well-balanced groups of *workers* in terms of number and of reputation. When executing this function *primaries* send messages to each others indicating the number of nodes they handle. Once all the messages received, *primaries* known the size T of the minimal group of *workers*. Each *primary* keep T *workers* under its responsibility and ask the random distribution of the other *workers* it handles.

---

reset

send (SIZE, $|P_i.P| + |P_i.A|$) to all others *primaries*

upon reception of (SIZE, $|P_i.P| + |P_i.A|$) from P_i on P_j

send (SIZE, $|P_j.P| + |P_j.A|$) to all others *primaries*

upon reception of $|P| - 1$ messages of type (SIZE, $|P_j.P| + |P_j.A|$) on P_i

input : set MSG of $|P| - 1$ messages of type (SIZE, $|P_j.P| + |P_j.A|$)

min ← $|P_i.P| + |P_i.A|$
for *i* ← 1 **to** |MSG| **do**
if $MSG_i.(|P_j.P| + |P_j.A|) <$ min **then**
min ← $MSG_i.(|P_j.P| + |P_j.A|)$
end if
end for

P_i.churn = 0
nToReset ← $|P_i.P| + |P_i.A|$ - min
send RESET to nToReset nodes in P_i.P, P_i.A and suppress them from those set
/* if we send a RESET to an actif node, we delete it from the actif set and put it to a temporary set (P_i.tmpSet). Since we get the result of the request, we suppress it from P_i.tmpSet */

upon reception of RESET from P_j on W_i

input : identity FP of the first-primary
/* if the node that receive this RESET, is executing a request, it finishes its execution and sends the response to his *primary* then it executes this routine */

send (RESET, W_i) to FP
receive ACKRESET from P_k
W_i.primary = P_k

upon reception of (RESET, W_i) from W_i on FP

input : Set P of *primary*

val ← random value in [1, |P|]
send (RESET, W_i) to P_{val}

upon reception of (RESET, W_i) from FP on P_k

send ACKRESET to W_i
$P_k.P = P_k.P \cup \{W_i\}$

Note:

This *reset* function has a quadratic cost if we consider the number of messages. It is thus, a costly function. However, this function will be rarely called if x (the number that trigger the execution of this function) Is well dimensioned.

III.2.b. Assignment considering reputations

In this strategy, *primaries* handle a range of reputation. For instance a *primary* can have under its responsibility nodes whom reputation is between 98 and 100 whereas an other *primary* handles nodes whom reputation is between 96 and 98... To have groups quiet balanced in terms of number of *workers*, multiple *primaries* can handle the same range of reputation.

When a *worker* join the system, there are several strategies. We could decide not to trust the new *worker* (it is the pessimistic strategy: we consider that all the nodes can be byzantine). In this case the new *worker* is assigned to the *primary* responsible for the nodes of bad reputation. Otherwise we could decide to trust the new *worker* and in this case the new node is assigned to the *primary* responsible for the nodes of good reputation (this is the optimistic strategy: very few nodes are likely to be byzantine). Or we could put the new *workers* with the *primary* in charge of the nodes of intermediate reputation.

It is better to consider the first solution (the pessimistic one). In fact, we don't know the nodes joining the system. If we give a high reputation to these new nodes, and it turns out they are byzantine, it is possible that the system will return incorrect result to clients. In fact, when we consider nodes of high reputations the system request few nodes to achieve a certain threshold of reliability for the result. So it is possible to request for instance only 3 *workers*. Suppose that there are three *workers* that join the system, that these three nodes are byzantine. Like we use the optimistic strategy, the *first-primary* assign them to the same *primary* which handles nodes of high reputation. If the *primary* request these three nodes and they collude to return a bad answer, the *primary* won't see that the result is false and will return this answer to the client. So we prefer to put the new nodes under the responsibility of the *primary* that is in charge of bad reputation. Like that if a byzantine node is in the system, it has to give lots of good answer before having a high reputation. And also, it is probable that the byzantine nodes will give some bad answers before having a high reputation, and so the system will detect them quiet soon.

---

arrival of a new node

init : node W_i wants to join the system
input : identity FP of the first-primary

send (JOIN, W_i) to FP
receive ACKJOIN from P_i
W_i.primary = P_i

upon reception of (JOIN, W_i) from W_i on FP
input : Set P of *primaries*

send (JOIN, W_i) to P_i where P_i is the primary responsible for the nodes of lowest reputation

upon reception of (JOIN, W_i) from FP on P_i

send ACKJOIN to W_i
P_i.P = P_i.P U {W_i}

To leave the system or when a *primary* notices the crash of one of its *workers*, the technique employed is the following:

upon leaving (or crash detection) of a node

init : node W_i wants to leave the system
send LEAVE to W_i.primary

upon reception of LEAVE from W_i or his crash detection on P_i
if $W_i \in P_i.P$ **then**
$P_i.P = P_i.P \setminus \{W_i\}$
else
$P_i.A = P_i.A \setminus \{W_i\}$
end if

In this strategy we don't need to use the field P_i.churn, unlike the random strategy. In fact, we don't search to obtain balanced groups, but we want groups where nodes have similar reputations.

With this strategy we have to assign again nodes during the execution in a dynamic way. Indeed, each *primary* handle a certain range of reputation, and nodes see their reputation evolved during execution, so they have to change of *primary* too. Each *primary* after the update of the reputations of nodes it handles (see section III.5 talking about the update of reputations) check that the nodes it handles have still a reputation included in the range of reputation it has in charge. If it isn't the cas the node has to change its *primary.*

dynamic change of goup executed by the *primary* P_i
input : set W of nodes being requested (i.e. nodes which reputation value has changed)
identity FP of the first-primary

for $k \leftarrow 1$ **to** |W| **do**
if W_k.reputation > P_i.maxR **or** W_k.reputation <= P_i.minR **then**
send (CHANGE, W_k) to FP
$P_i.P = P_i.P \setminus \{W_k\}$
end if
end for

Upon reception of (CHANGE, W_k) from P_i on FP
input : Set P of *primaries*

send (CHANGE, W_k) to P_j where P_j is the primary responsible for the nodes of reputation in [P_j.minR, P_j.maxR[, and $w_k \in [P_j.\text{minR}, P_j.\text{maxR}[$

Upon reception of (CHANGE, W_k) from FP on P_j
$P_j.P = P_j.P \cup \{W_k\}$
send ACKCHANGE to W_k

upon recpetion of ACKCHANGE from P_j on node W_k

W_k.primary = P_j

Note:

if we consider that lots of nodes have to change their *primary*, this method is expensive. So, in this case, we could wait a certain time, or that a fixed number of nodes have to change their *primary* to really do these modifications. This would permit to send less messages. In fact, a *primary* can regroup in only one message all the identities of nodes that have to move towards the same *primary*. We will use the field P_i.NotInF of *primaries* to stock the set of nodes whom reputation doesn't correspond anymore to the range of reputations handle by the *primary*.

Notes about the two strategies presented:

- The network is supposed partially synchronous and the *primaries* are assumed reliable, thus these two strategies insures that eventually a node which want to join or leave the system would be able to do so.

- We could have some problems of well-balanced nodes with this two strategy. In fact, it is possible that a *primary* hasn't enough nodes under is responsibility to be able to answer the request of clients. Or conversely, it is possible that a *primary* handles too much nodes and that its computation are slower than if it has less *workers*. So we need a *load balancing* mechanism to maintain groups with quiet enough number of nodes to execute request in a fast way.
  When a *primary* won't have enough nodes to answer the clients, it will call the *load balancing* function. This function deactivate the *primary* putting its inPool boolean to false. The *workers* under the responsibility of this deactivated *primary* will be redistributed on the active *primaries* (considering the reputations if the strategy is in function of the reputations). Similarly if a *primary* is in charge of too much *workers* the system redistribute the overload over the others *primaries*.

In function of the strategy chosen (random or in function of the reputations) the *load balancing* function is different.

Below in black there is the pseudo-code of the random *load-balancing* function. The writings in blue/green corresponds to the code we have to add for the *load-balancing* function when we consider the reputations of *workers*.

---

load balancing in case of overload

upon the number of workers on primary P_i is too high *

input: identity FP of the first-primary

the list list_workers of the half of the workers of P_i. Those *workers* correspond to the highest nodes in terms of reputation owned by P_i

if (P_i.able_to_send_division == 1) **then**

value = min {w_i.reputation \ $w_i \in$ list_workers }

send <DIVISION, value, P_i.maxR> to FP

wait message from FP

end if

upon reception of <DIVISION, value, P_i .maxR**> from P_i on FP**
if |FP.PI| > 0 **then**
FP.PI = FP.PI \ {head(FP.PI)}
FP.PA = FP.PA U {head(FP.PI)}
update the maxR of P_i to value in FP.PA
update the maxR of (head(FP.PA)) to P_i .maxR
update the minR of (head(FP.PA)) to value
if (|FP.PA| == 2) {
broadcast ABLE_TO_FUSION to all *primaries* in FP.PA
}
// we send the number of the primary newly introduce in the system to P_i
send <ack_DIVISION, head(FP.PI).identity> to P_i
else
send <unack_DIVISION> to P_i
end if

upon reception of message from FP on P_i
input: the list list_workers of the half of the workers of P_i . Those *workers* correspond to the highest nodes in terms of reputation owned by P_i

if message == <ack_DIVISION, numPrimary> **then**
value = min {w_i .reputation \ $w_i \in$ list_workers }
send <GIVE_WORKERS, list_workers, value, P_i.maxR> to numPrimary
P_i .P = P_i .P \ {list_workers}
P_i .maxR = value
else if message == <unack_DIVISION>
P_i .able_to_send_division = -1
else // message == <ABLE_TO_FUSION>
P_i.able_to_send_fusion = 1
end if

upon reception of (GIVE_WORKERS, list_workers, value, P_i.maxR) from P_i on P_k
P_k.minR = value
P_k.maxR = P_i.maxR
for *i* ← 1 **to** |list_workers| **do**
send NEW_PRIMARY to list_workers$_i$
end for

upon reception of NEW_PRIMARY on W_j from P_k
W_j.primary = P_k

---

* we will define experimentally what is too high

The *workers* are potentially byzantine, so it is possible that they don't execute correctly the phase “upon reception of NEW_PRIMARY”. However as our *primaries* are correct the primary named P_i in the pseudo-code won't after the execution of the *load balancing* function consider the worker W_j under its responsibility. P_k will consider the worker under its charge, and after if the *worker* W_j decide to not answer P_k, P_k will just consider it has crashed and will suppress it from the system.

So the fact, that some *workers* don't execute the procedure correctly doesn't corrupt the system.

---

load balancing in case of underload

upon $|P_i.PA| == 0$ and $|P_i.PI|$ doesn't permit to form groups on P_i
input: identity FP of the first-primary

```
if P_i.able_to_send_fusion == 1 then
        send <FUSION> to FP
        wait message from FP
end if
```

upon reception of <FUSION, $|P_i.P|$> from P_i on FP

```
if (|FP.PA| > 1) then    // we necessarily want at least a primary in the system
        FP.PA = FP.PA \ {P_i}
        FP.PI = PF.PI U {P_i}
        if |FP.PI| > 1 then
                broadcast ABLE_TO_DIVISE to all the primaries in FP.PA
        end if
        send <ack_FUSION> to P_i
        wait <WORKERS_TO_FUSE, list_workers> from P_i
else
        send <unack_FUSION> to P_i
end if
```

upon reception of a message from FP on P_i
input: identity FP of the first-primary

```
if message == <unack_FUSION> then
        P_i.able_to_send_fusion = -1
else if message == <ack_FUSION> then
        send <WORKERS_TO_FUSE, P_i.P> to FP
else    // message == <ABLE_TO_DIVISE>
        P_i.able_to_send_division = 1
end if
```

upon reception of <WORKERS_TO_FUSE, list_workers> from P_i on FP

```
limit ← 0.0

for i ← 1 to |list_workers| do
```

if the group formation strategy corresponds to the random one **then**
chose randomly a *primary* on FP.PA
else // the group formation strategy is function of the reputations of *workers*
chose the *primary* P_k in FP.PA such that P_k.minR <= list_workers$_i$ < P_k.maxR
if such a *primary* doesn't exist **then**
find a *primary* P_m such that P_m.maxR is the closest of P_i.minR
find a *primary* P_n such that P_n.minR is the closest of P_i.maxR
$\text{limit} = \frac{P_i.\text{maxR} - P_i.\text{minR}}{2} + P_i.\text{minR}$
send <LIMIT_MAX, limit> to P_m
send <LIMIT_MIN, limit> to P_n
chose the *primary* P_k in FP.PA such that:
P_k.minR <= list_workers$_i$ < P_k.maxR
end if
end if
send <WORKER_FUSED, list_workers$_i$> to P_k
end for

upon reception of <WORKER_FUSED, list_workers$_i$> from FP on P_k
P_k.P = P_k.P U {list_workers$_i$}

upon reception of <LIMIT_MAX, limit> from FP on P_k
P_k.maxR = limit

upon reception of <LIMIT_MIN, limit> from FP on P_k
P_k.minR = limit

---

* we will define the value of x experimentally

III.3. Group formation strategies

Each *primary* perform group formation. This means that each *primary* form subset of *workers* beyond the nodes it handles. To from this subset multiple strategies are possible: *fixed-fit*, *first-fit*, *tight-fit*, *random-fit* and the method of Arantes et al. []. The *primary* submit the same task to all the *workers* belonging to the same subset.

The four firsts methods presented corresponds to strategy used in the article written by Sonnek et al. [] (which is a centralized probabilistic algorithm). The last method corresponds to the method presented in the article of Arantes et al. []. Our hierarchical solution is implemented above this two centralized algorithms (presented during the section IV).

III.3.a. Fixed fit

With the fixed-fit strategy groups of *workers* are formed without taking into account their

reputations.

The number *nb* of machines contained in each groups formed is fixed statically (by an administrator for instance). The *primary* chose randomly *nb workers* to form a subset.

Note:

When *workers* are assigned over *primaries* in function of their reputations (see section III.2.b.) , the groups formed with the *fixed-fit* strategy by a *primary* have similar reputations.

III.3.b. First-fit

In the article presented by Sonnek et al. the *first-fit*, *tight-fit* and *random-fit* strategy permits to form only odd groups.

The first-fit group formation strategy takes into account the reputations of nodes to create subsets of *workers*. Nodes are added to a group until it achieves a certain threshold $\lambda target$ of reliability asked by clients (when a client sends a request it adds the reliability he wants for the result in the message sent).

In the *first-fit* strategy the *workers* are ordered in descending order of reputations. The *primary* regroups the *workers* (following the order of the list previously ordered) until achieving a the threshold $\lambda target$, or until a group contained the maximum number of nodes authorized in a group. Thus in the worst case, thanks to this maximum number of nodes this method corresponds to the *fixed-fit* strategy. There is a minimal number of *workers* a minimal number of *workers* authorized in each groups too.

This algorithm corresponds to the one describe in the pseudo-code below:

---

first-fit executed by the *primary* P_i

input : threshold $\lambda target$ to reach
value Gmin of the minimal number of *workers* authorised in a group
value Gmax of the maximal number of *workers* authorised in a group

```
G ← sorted list of Pi.P according to reputation of nodes
while |G| ≥ Gmin do
    repeat
        Assign the most reliable worker Wr from G to Gi
        G ← G \ {Wr}
        if |Gi| ≥ Gmin then
            update λi
        end if
    until ((λi ≥ λtarget and |Gi| ≥ Gmin) or |Gi| == Gmax)
end while
```

The computation of λ_i is done thanks to the formula:

$$\sum_{m=k+1}^{2k+1} \binom{2k+1}{m} * \prod_{i=1}^{2k+1} r_i^{\alpha_m} * (1-r_i)^{1-\alpha_m} \quad \text{where} \quad \alpha_m = \frac{\binom{2k}{m-1}}{\binom{2k+1}{m}}$$

2k+1 corresponds to the size of the group. This formula computes the probability that the majority of the *workers* chosen to form the groups are correct. In Sonnek et al. the result returned to the clients is the result that is in majority beyond all the results returned, that is why the formula search the probability to obtain a majority answer.

Note:

In the case where the assignment of *workers* over *primaries* is done considering the reputations, compute the *first-fit* strategy corresponds to form group in a random way (see section III.3.d for the description of the random formation group). In fact as a *primary* handles *workers* with similar reputations isn't mandatory to execute the preliminary sort of the nodes. We will check that experimentally.

III.3.c Tight-fit

This strategy looks like the previous one. However, nodes are grouped in an optimal way. Indeed, this method to form groups whom reliability is the closest from the threshold $\lambda target$.

This strategy is surely more expensive that the *first-fit* strategy, but it permits to obtain more groups. And the groups formed are more balanced in terms of reliability.

The pseudo-code of this strategy is the following one:

---

tight-fit executed by the *primary* P_i

input : threshold $\lambda target$ to reach
value Gmin of the minimal number of *workers* authorised in a group
value Gmax of the maximal number of *workers* authorised in a group

$G \leftarrow$ sorted list of $P_i.P$ according to reputation of *workers*
while $|G| \geq Gmin$ **do**
Assign the most reliable *worker* W_r from G to G_r
$G \leftarrow G \setminus \{W_r\}$
repeat
use binary search to identify the smallest set G_r of *n workers* from G such that λG_r exceeds $\lambda target$ minimally ($Gmax \geq n \geq Gmin$)
$G \leftarrow G \setminus G_r$
if $|G_r| \geq Gmin$ **then**
update λ_r
end if
until (($\lambda_r \geq \lambda target$ **and** $|G_r| \geq Gmin$) **or** $|G_r| == Gmax$)
end while

Like we said when using Sonnek et al. group formation group strategy we want to form odd groups. When using “binary search to identify the smallest set G_r of *n workers* from G” we search two workers. These two workers are side by side in the descending order of reputations and their reputations are above 50.

The following example help explaining the principle:
Assumed we have got 14 workers with the reputations above:

Number in the descending order list of workers	0	1	2	3	4	5	6	7	8	9	10	11	12	13
reputations	90	90	90	80	80	80	70	70	70	60	60	30	30	30

The two instructions Assign the most reliable *worker* W_r from G to G_r; $G \leftarrow G \setminus \{W_r\}$ forced the worker 0 to be added to the group.

Then, workers 9 and 10 would be added (the least reliable pair with reliability > 0.5). If the LOC for this group is greater than the target LOC, then we're done. If not, we remove pair 9/10, and use binary search to find the next pair (half the interval between 9/10 and 0). In this case, we'd add worker pair 4/5, so the group would be {0, 4, 5}. If the LOC for this group is greater than the target LOC, then we're done. If not, we try another pair; we halve the interval between the last attempt (4/5) and the head of the list; in this case, we'll try 2/3 ....

If we end up with the group {0, 1, 2} and the group LOC is still not greater than the target LOC, then we add more workers to the group using the same process. So, we'd first add 9/10 to the group {0, 1, 2}, check the LOC for the group, etc.

III.3.d. Random-fit

In this strategy the nodes are added in groups in a random way and considering their reputations. The *primary* adds nodes in a group until achieving a the threshold $\lambda target$ (and the minimal number of nodes in a group), or until the group contained the maximum number of nodes authorized in a group.

This strategy is different from the *first-fit* strategy because the *workers* are added in a random way in a group and not considering the descending order of reputations of nodes.

III.3.e. Arantes et al. Method []

In the method proposed by Arantes et al. we choose nodes respecting $P_B / P_C \leq th$ to form groups. P_B corresponds to the probability that all the nodes selected to form the groups are byzantine. P_C corresponds to the probability that all the nodes in the group formed are correct. *th* is the threshold we want to achieve.

In the article it isn't precise how to form this group. To obtain the minimal group it woiuld be necessary to build all the possible groups of workers, compute for each groups P_B and P_C, and chose the minimal one. These technique is too expensive. So we will use the strategies used in the *first-fit, tight-fit* and *in the random-fit* strategy to form the groups. The modification will only resides in the way to calculate λ_i. If we chose the Arantes et al. method we will compute λ_i the thanks to P_B / P_C. Whereas in the others methods present previously the λ_i is compute thanks to the formula presented in the section III.3.b. When using the tight-fit with Arantes et al. we don't need to search two workers thanks to the use of binary search but only one.

When using the Arantes and al. method we can form odd as even groups.

III.4. Additional replication strategies

The goal of probabilistic method is to request the minimum number of machines to be able to return a reliable result to the client. As we saw in the previous section *primaries* form groups and request a whole group with the same request. And as we said too, our hierarchical solution is based upon centralized existing solutions (Sonnek et al. and Arantes et al.). In Arantes et al. if all the nodes requested give the same answer, the result is returned to the client. However if all the nodes don't return the same answer, this means that the threshold of reliability for the answer isn't achieve. So we need to replicate the task on additional *workers* until achieving this threshold. In Sonnek et al. They only want a majority answer to return the result. If the majority isn't achieve they just send fail to the client.

So in this section we will detail the replication done in Arantes et al. We will also detail the *progressive* and *iterative redundancy*, that are describe in the article [] and that we will implement when we will use the *fixed-fit* strategy.

III.4.a. Until achieving a certain threshold

This section describes the strategy used in Arantes et al. []. When the nodes don't give the same result we have to request others nodes to obtain the threshold wanted by the client. Those additional nodes are chosen considering their reputations. The additional nodes has to be different from the nodes already requested for the task.

This pseudo-code of the algorithm is the one above:

---

additional replication

input: the set R of nodes that was previously requested whom the *workers* don't send the same answer
the set R_1 of nodes that returned the most frequent* value
threshold $\lambda target$ to reach

choose a set R_2 (thanks to *first-fit*, *tight-fit* or *random-fit*) such that $R \cap R_2 = \emptyset$ and formula2 > (1 - $\lambda target$)

computation of formula2

input: the set $R \cup R_2$ corresponding to all the nodes request for the task considering
the set $R_1 \cup R_2$ corresponding to the set of nodes supposed to returned the same answer

output: the value corresponding to the probability that the value returned by

return $\dfrac{P_{SC}}{\left(\prod_{j \in R_1 \cup R_2} (1 - reputation_j)\right) + P_{sc}}$

where $P_{sc} = \left(\prod_{j \in R_1 \cup R_2} reputation_j\right) * \left(\prod_{j \in (R \cup R_2) \setminus (R_1 \cup R_2)} 1 - reputation_j\right)$

---

* the most frequent value corresponds to the value that obtained the greatest value when computing

formula 2.

If all the additional nodes requested return the same value that the most frequent value (this means that we obtained a subset of *workers* in the group formed that permits the formula2 to be less than (1 - *λtarget*)) then we can returned the result to the client. If not we have to compute again the additional replication function.

III.4.b. Progressive reudundancy

In the general way, to tolerate the byzantine faults we replicate a task over k *workers* and then we returned to the client the majority answer ($\lfloor \frac{k}{2} \rfloor + 1$). The *progressive redundancy* is an optimistic method used to reduce the number of *workers* needed. When we use this technique we first request only $\lfloor \frac{k}{2} \rfloor + 1$ *workers*. If all the nodes give the same answer then a consensus is achieved and the answer is returned to the client. If this isn't the case, we have to replicate the task on the minimal number of nodes permitting to obtain this consensus. This process is executed until this consensus is effective.

For instance, if k=7, first we will replicate the task on 4 *workers*. If the 4 nodes don't agree on the result, for example 3 give the same answer and the other give an other answer, we have to replicate the task on an other node. We have to replicate until there is a majority answer agreed by at least 4 nodes (whatever the number of nodes requested eventually, if there are 100 nodes requested and that there is a majority answer returned by 4 nodes, it this answer that we will returned to the client, we just take into account the initial value of k, not the final number of nodes requested to achieve the consensus). If k=7 and that 8 nodes has already been requested and that 4 returned a result and 4 others an other result, even if a consensus has been achieved, we have to replicate to decide what is the final result.

III.4.c. Iterative redundancy

In this strategy the primary distribute tasks to nodes in order to achieve a certain threshold of reliability. However, the authors that introduced the *iterative redundancy* [] shown that this technique doesn't necessitate to know the reputation of nodes. They assume that all the machines have the same reputation, and say that in this case, it exists a computation indicating the number *d* of *workers* needed to achieve a certain threshold of trust. The authors consider the worst case where all byzantine node always enter in collusion. So only two results are possible for one task, the correct one and the incorrect one. They show that we only need to obtain a difference between the correct and incorrect results of *d* to achieve the reliability threshold wanted.

So, the replication will be done until having *x* machines giving the bad result and *x+d* giving the good.

As we indicated this technique consider that all nodes have the same reputations. So like in our system nodes see their reputations evolved, we can't use this method. However, we could analyze the behavior of this technique in our system despite the reputations evolution.

III.5 reputations update strategies

The *primary* has to update the reputations of the nodes it has requested lately.

Before being able to update the reputations of nodes, *primaries* have to determine what is the assumed correct answer. The *workers* that return that response will see their reputations increased, whereas the others *workers* will have their reputations decreased.

We need an algorithm able to detect the majority answer. Moreover, this algorithm has to indicate if we could return the result to the client or if we need to do some additional replications. To do that the algorithm maintains a structure *L* which contains the set of the answers returned by the requested *workers*. Each of this answer will be weighted by the number of *workers* that have returned the answer considered. The algorithm returns a quintuplet: the structure *L*, one of the majority answer *A*, the number *x* of *workers* that returned the answer *A*, the number *nb* of answers that obtained the same number of *workers* than *A* and the number *tot* corresponding to the total number of *workers* requested for the task. We return *L* because the first time the algorithm is called *L* is empty and it is the algorithm that fill this structure.

If our hierarchical solution is based on Sonnek et al. algorithm we will return *A* to the client if *nb*=1 and that $x \geq \lfloor \frac{tot}{2} \rfloor + 1$. In this case the *workers* that returned *A* have their reputations increased and the *workers* that didn't return *A* have their reputations decreased. if *nb*=1 but $x < \lfloor \frac{tot}{2} \rfloor + 1$ then we will send fail to the client but we will increase the reputation of the *x workers* and decrease those of the others *workers*. If none of the conditions enunciated previously is checked then the reputation of all the *workers* requested for the task will see their reputations decreased.

If our solution is under Arantes et al. algorithm the majority answer doesn't correspond to the answer returned by the biggest number of *workers* beyond those requested, but corresponds to the answer having the biggest value of formula2. In this case *x* will correspond to the value returned by formula2. If *nb*=1 and (*x = tot* or $x < (1 - \lambda target)$) then we return *A* to the client and increased the reputations of *workers* that returned *A*, the others *workers* will have their reputations decreased. If the conditions aren't checked we have to replicate and wait to have the condition checked to update the reputations of the *workers*.

Below we put the pseudo-code of the algorithm described. We write it as if we where using Sonnek et al. but it is quiet the same algorithm used when we employed Arantes et al. (just change x by the computation of formula2).

---

find the majoritary answer

---

input : set S of nodes previously requested for a specific request

set L of (A, x) where A is an answer given by a *worker*, and x is the number of *workers* that give that answer for the request

number tot corresponding to the number of *workers* already requested for the request

// the fisrt time this function is called for a request the set L is empty and tot equals 0

output : (L, A, x, nb, tot) where L is the structure defined above, A is the majority answer, x is the number of *workers* which give this answer, nb the number of answer returned by x *workers* too, and tot is the total number of *workers* requested

max ←0

for *i* ← 1 **to** |S| **do**
    if S_i.a belongs to L **then**
        remove (S_i.a, x) from L
        add (S_i.a, x+1) to L
        if max < x+1 **then**
            max = x+1
            nb = 1
        else if max == x+1 **then**
            nb++
        end if
    else
        add (S_i.a, 1) to L
        if max < 1 **then**
            max = 1;
            nb = 1;
        else if max == 1 **then**
            nb++
        end if
    end if
end for
tot ← tot + |S|

(A, x) ← $(S_i.a, x) \in L$ where $x = max\{y \in \mathbb{N} \setminus \forall j, (S_j.a, y) \in L\}$

return (L, A, x, nb, tot)

We now present the different strategy that we will use to update the reputations in our solution which are the *symmetrical*, *asymmetrical* and the update presented in the article written by Sonnek et al.

III.5.a. Symmetrical

The *symmetrical* strategy permits to increased or decreased the reputations thanks to a percentage fixed statically.

---

symmetrical update of reputations

input : set S of nodes previously requested
    value v of the "correct" answer
    value Rmin of the minimal reputation authorised
    value Rmax of the maximal reputation authorised
    value x percentage used to modify reputations
    value y percentage used to modify reputations

for *i* ← 1 **to** |S| **do**
    if S_i.a == v **then**
        S_i.reputation = S_i.reputation + S_i.reputation * x

```
            if Si.reputation > Rmax then
                Si.reputation = Rmax
            end if
        else
            Si.reputation =  Si.reputation -  Si.reputation * y
            if Si.reputation < Rmin then
                Si.reputation = Rmin
            end if
        end if
    end for
```

In the *symmetrical* strategy the value of x and y are identical.

III.5.b. Asymmetrical

The *asymmetrical* strategy is similar to the *symmetrical* one, except that in the former the value of x and y are different.

It is usual in *asymmetrical* strategy to take y > x. In this way, a node will obtain the trust of the system slower than it will lost it. In case of a bad answer, the application will penalize strongly the node.

III.5.c. Sonnek et al. [] update of reputations method

The strategy used by Sonnek et al. consists to count the number of assumed correct answer provided by a node, and the number of request that have been submitted to it. The reputation of the node is then compute thanks to the formula: S_i.reputation = (S_i.totC + 1) / (S_i.totR + 2).

So the reputations of nodes is initialized to ½, and at the end of each task executed the reputation is computed again thanks to the formula.

III.6. Complements

Regardless of the strategy employed, there are two transversal strategies that can be used to avoid byzantine nodes to corrupt the system.

When the reputation of the nodes is too low, we lost in latency. In fact, if we use a group formation strategy based on the reputations of nodes, we request more nodes. We have to wait the answers of those nodes or to detect their failures, so this is slower than if we request less nodes. Moreover as the nodes have bad reputation, it is quiet probable that we have to do some additional replication to obtain the answer to return to the client. So the latency is increased. To avoid this we could use a blacklist strategy. This means that we can suppress from the system nodes which have their reputations behind a certain threshold. For instance we could suppress all the nodes that have their reputations behind 50.

Moreover, some byzantine nodes give during a time good answers in order to have their reputations increased. Once they have a good reputation, they can send bad answer to the system without necessarily getting caught by the system. To avoid this situation we could regularly reset the reputation of all the nodes of the system to an initial value. If the assignment of *workers* over

primaries depend of the reputations, nodes have to change their *primary* when the reset of reputation take place.

We will make experimental comparison between systems that have those mechanisms of blacklist and reset of reputations, and systems without these techniques. This will permit to know if these mechanisms are really efficient to avoid byzantine faults.

III.7. primaries interrogation (by the clients) strategies

Clients send their requests to the *first-primary.* This last will then choose the primary to which it will forward the request.

Several strategies are possible: the client can request in a random way one of the *primaries*, it can request too a *primary* depending on the cost it is ready to pay.

III.7.a. Random

This strategy permits to clients to request in a random way any *primary.*

---

random requesting of *primary*

input : identity FP of the first-primary
request r

send (REQUEST, r) to FP
wait for the answer

upon reception of (REQUEST, r) from client$_i$ on FP
input : Set P of *primary*

val ← random value in [1, |P|]
send (REQUEST, r, client$_i$) to P$_{val}$

This strategy permits to obtain balanced charge on *primaries* in terms of the number of requests received.

III.7.b. Strategy considering the cost

This strategy is mostly adapted when the assignment of *workers* on *primaries* is done considering their reputations.

Indeed, when *primaries* are divided in range of reputation, some *primaries* are responsible of *workers* of good reputation. These *primaries* have to request less nodes than the *primaries* in charge of the *workers* of bad reputations if the formation group strategy chosen is *first-fit, tight-fit, random-fit,* or *Arantes et al.* Considering the number of nodes the client want to request and the reputations of these machines, the client will call the *primary* responsible for machines of high, intermediate and low reputations.

If we want the *first-primary* to stay a serving hatch the client has to precise with its request

the reputation of the machines it wants to request. Otherwise the client might indicate the number of machines it wants to request, and with that number the *first-primary* could calculate to which *primary* it has to forward the request.

We focus on the method where the *first-primary* stay a serving hatch (permitting our system to stay scalable and insure a good throughput to treat requests):

---

requesting of *primary* according to the reputation it handles

input : identity FP of the first-primary
request r
reputation R of the machines requested

send (REQUEST, r, R) to FP
wait for the answer

upon reception of (REQUEST, r, R) from client$_i$ on FP
input : Set P of *primary*

send (REQUEST, r, client$_i$) to P_i where P_i is the primary responsible for the nodes of reputation in [P_i.minR, P_i.maxR[, and $R \in [P_i.minR, P_i.maxR[$

IV. Centralized algorithms serving to comparison with our solution

The algorithms with which we compare our hierarchical solution are centralized probabilistic algorithms. They have only one *primary* to handle all the requests.

These algorithms don't have to implement the assignment of *workers* to *primaries* strategies neither the *primaries* interrogation (by clients) strategies. In fact, all *workers* are handled by only one *primary* and clients request this only node.

So the centralized algorithms just implement some formation groups strategies, some additional replication strategies, and some update reputations strategies.

We describe the strategies used by the two algorithms with which we do our comparison.

IV.1. Strategies used by Sonnek et al.

In the centralized algorithm of Sonnek et al. the assumptions that there is no collusion is put. This means that byzantine nodes can't agree on the same wrong result and return it to the *primary.*

To form groups they use the *fiexed-fit*, *first-fit*, *tight-fit* and *random-fit* strategy describe in section III.3.a, III.3.b, III.3.c, and III.3.d.

The update reputations strategies correspond to the strategy presented in the section III.5.c., where the value is initialized to ½ and then it is modify depending on the number of the number of assumed correct answer return by the node and depending on the total number of request it has received.

In the article they present they don't consider additional replication. In Sonnek et al. if an absolute majority isn't obtain we send “fail” to the client, indicating him that if he want an answer it has to send again his request to the *primary.*

In Sonnek et al. they formed groups considering the reputations of nodes (except if the formation group chosen is the *fixed-fit* strategy). After to send an answer they wait that a majority of nodes agree on the same result to returned it to the client. Here we could thinks that there is no sense to create groups thanks to reputation and then wait for a majority without taking into account the reputation of the nodes in this majority. Indeed, if we form a groups of 5 nodes, if 3 of them returned a result x, and that the subset formed by these 3 *workers* have a reputation of 0.3, if the two others nodes give the same result *y* and they have together a reputation of 0.65. Intuitively we want to send *y* to the client. However when processing Sonnek et al. the result send is *x.* They done this because they consider the reputation as an estimation of the reliability but also of the speed of execution. If a node isn't byzantine but is slow its reputation will be low.

IV.2. strategies used by Arantes et al.

In this algorithm the byzantine nodes are authorized to collude.

The formation group strategy used in Arantes et al. corresponds to the one describe in section III.3.e.

In this algorithm the *primary* isn't authorized to send “fail” to the client, so there are the mechanisms of additional replications presented in section III.4.a. that are implemented to permit to return an assumed correct answer to the client.

Several reputation strategy are presented in the article written by Arantes et al.: the symmetrical and the asymmetrical one, and also the one used in BOINC.

In BOINC (Berkley Open Infrastructure for Network Computing which is a platform where using *workers* in a *Desktop Grid* to replicate tasks, as us) the reputations of nodes are computes thanks to the error rate of the nodes. The formula to update the reputation of one node is: S_i.reputation = $1 - (S_i.totR - S_i.totC) / S_i.totR$). With totR the total number of requests send to S_i, and totC the number of answers assumed correct returned by this node.

During our experimental tests we will first compare the two centralized strategies, there the BOINC reputation update strategy will be used. But when we will analyze our hierarchical solution working under Arantes et al., we will never use the BOINC reputation update. In fact, as we said before, we want to initialize new nodes with low reputation. But if we use BOINC, we will be forced to give new nodes a reputation of 1.

V. Implementation

There are three options to test code: doing real tests on real platforms, or using an emulator or using a simulator.

Doing tests on real platforms is really complex. In fact, the execution isn't reproducible because of the dynamism of the machines. To an execution from an other the machines available

aren't the same. It is impossible to impose some properties on a real system like the number of machines wanted, those which are byzantine. Similarly the messages aren't send/receive in the same order from one run to an other. So to debug applications we can't use real environment.

Emulators permit to use only one machine to represent multiple machines. However, results aren't reproducible, and we absolutely want this characteristic to be able to debug.

So we use a simulator. A simulator permits the reproducibility of results. It makes too the discretization of the time, permitting to execute big execution in a short time.

V.1. SimGrid

We have chose to simulate Sonnek et al., Arantes et al. and our solution thanks to SimGrid version 3.11 []. In this section we will describe SimGrid and then we will indicate what are the actions to follow to install SimGrid.

V.1.a. description

We choose SimGrid to simulate our applications because it is able to simulate large-scale applications in a short time. It also permits to use networks models more and less realistic. So to model, and analyze in an algorithmic point of view we could first use a constant network where all the messages take the same unit of time. Then it is possible to deploy a realistic network parsing some network traces and giving them to SimGrid, to see the probable behavior of the system in real platform.

Interfaces of SimGrid used:

SimGrid offers multiple interfaces. We use MSG and XBT.

MSG is an user interface to facilitate the development of distributed applications where processes has to execute tasks.

There are 4 abstractions done by this interface: the agent, tasks, host, and mailbox. The agent corresponds to the data and the code executed on a host, the tasks. A host corresponds to a physical machine we want to simulate. Mailboxes correspond to the boxes of an host, these box permits to the host to send and receive messages

XBT is a C user interface providing data structure like FIFO, dynamic array...

Platforms and deployment files:

It is possible to give to Simgrid 2 xml files: one to describe the platform and one to describe the deployment. This isn't mandatory to use this 2 files, in fact, it is possible to program the platform and the deployment using functions provided by the MSG interface.

The file describing the platform contains all the name of the hosts we want to be part of our system, the links and route between hosts.

For each host it is mandatory to define its identity and its power which represents the maximum value of flop/s the CPU is able to manage. There are others options possible like define the number of core the CPU has.

For each links are represented by an identity, a bandwith in bytes per second. It is optional, but possible to indicate the value of the latency of a link in seconds. It is possible to have multiple

flux sharing the same links (like it is done in TCP).

To describe a route this is mandatory to indicate the source and the destination (corresponding to host identity). It is possible to indicate that the route is the same from the source to the destination that the route from the destination to the source thanks to the option symmetrical. We have also to indicate all the links that constitute the route.

The deployment contains the description of the process. For each process we indicate the identity of the host on which it is executing. We also indicate the name of the function the process has to execute. Finally we could put some value we want the process to know and use.

V.1.b. Installation

To install SimGrid we need first to install cmake.

installation of Cmake:

In order that Cmake functions we need to have:

make
perl and libpcre
compilateur c and c++
cmake gui
cmake

To install Cmake we need then to do:

sudo ./bootstrap
sudo make
sudo make install

installation de SimGrid:

First we need to download SimGrid:
git clone git://scm.gforge.inria.fr/simgrid/simgrid.git simgrid

This permits to create a directory with the sources of SimGrid. Then to install SimGrid we have to execute the following steps:

cd simgrid
git checkout v3_11_x
cmake -Denable_maintainer_mode=on -DCMAKE_INSTALL_PREFIX=/home/marjo/simgrid ./
make
make install

V.2. details about the code

In this section we will describe briefly the files we created and we will present some the details about our implementation on SimGrid.

V.2.1. files used

We created 7 files and ...(to complete) headers associated to those:

primary.c corresponds to the code of a process waiting request from clients and join from *workers* (in the centralized algorithm), or request from the *first-primary* (in the hierarchical solution). It call the different strategies developed in function of what it is asked.

worker.c corresponds to the code of a process asking to join the system and waiting to execute the request. Each workers has an unique identity, this permits us to attributes to each one a unique trace taken on Failure Trace Archive []. These traces taken from real existing systems (like seti@home) indicate at what time a node has crashed and at what time it recovers. Thus, when a *primary* send a request to a *worker,* it has to check in its trace if it is always present on the system, or if it has crashed. It has also to check if it doesn't crash during the execution of the task. If the node has crashed, as we are in a simulator the *worker* simply send “crash” to its *primary.* Then its has to wait (thanks to a sleep) until the time where its trace indicate its recovery. After its recovery the *worker* must restart its execution from the beginning, this means it has to ask to join the system.

client.c corresponds to the code of a process requesting the *primary* (in the centralized algorithm) or the *first-primary* (in the hierarchical solution) waiting an answer or indication of the request has failed before sending an other request.

group_formation_strategy.c contains the implementation of the different group formation strategies evoked section III.3.

additional_replication_strategy.c contains the code of the different additional replication strategies evoked section III.4.

reputation_strategy.c contains the code of the different way to update the reputations evoked section III.5.

simulator.c corresponds to the launcher. It collects the informations passed in command line about the strategies used during the simulation. It reads the failure traces of *workers*, to know the minimal time where a *worker* was available in the system which produce those traces. In this way, we could start the simulation at this time. This file also contains the code permitting to take in place the platform and to run all the processes describe in the xml files (given in command line).

Références :

[1] L. Lamport, R. Shostack, and M. Pease. The Byzantine Genereals Problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382-401, 1982

[2] K. Driscoll, B. Hall, H. Sivencroma, and P. Zumsteg. Byzantine Fault Tolerance, from Theory to Reality, *22nd International Conference, SAFECOMP*, 2003

[3] R. Guerraoui and M. Yabandeh. Independant Faults in the Cloud, *LADIS'10*, 2010

[4] M. Castro and B. Liskov. Practical Byzantine Fault Tolerance, In *3rd Symp. on Operating Systems Design and Impl.*, February 1999

[5] R. Guerraoui, V. Quéma and M. Vukolić. The Next 700 BFT Protocols, *EPFL Technical Report*, 2009

[6] R. Kotla, L. Alvisi, M. Dahlin, A. Clement and E. Wong. Zyzzyva : Speculative Byzantine Fault Tolerance, *SOSP'07*, 2007

[7] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin and T. Riché. UpRight Cluster Services, *SOSP'09*, 2009

[8] J. Cowling, D. Myers, B. Liskov, R. Rodrigues and L. Shrira. HQ Replication : A Hybrid Quorum Protocol for Byzantine Fault Tolerance, *OSDI'06 : 7th USENIX Symposium on Operating Systems Design and Implementation*, 2006

[9] Y. Zhang, Z. Zheng, and M. Lyu. BFTCloud : A Byzantine Fault Tolerance Framework for Voluntary-Resource Cloud Computing, *IEEE Computer Society*, 2011

[10] J. Sonnek, A. Chandra, and Jon B. Weissman. Adaptative Reputation-Based Scheduling on Unreliable Distributed Infrastructures, *IEEE Computer Society*, 2007

[11] S. Duan, K. Levitt, H. Meling, S. Peisert, H. Zhang. ByzID : Byzantine Fault Tolerance from Intrusion Detection, *18th International Conference on Principles of Distributed Systems (OPODIS)*, 2014

[12] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. K. Reiter and J. J. Wylie. Fault-Scalable Byantine Fault-Tolerant Services, *SOSP'05*, 2005

[13] Y. Amir, B. Coan, J. Kirsch, J. Lane, and J. Hopkins. Byzantine Replication Under Attack, *IEEE/IFIP International Conference on Dependable Systems and Networks*, 2008

[14] J. Yin, J-P. Martin, A. Venkataramani, L. Alvisi and M. Dahlin. Separating Agreement from Execution for Byzantine Fault Tolerant Services, *SOSP'03*, 2003

[15] A. Agbaria and R. Friedman. A Replication- and Checkpoint-Based Approach for Anomaly-Based Intrusion Detection and Recovery, *ICDCSW'05*, 2005

[16] M. Correia, N. Ferreira Neves and P. Veríssimo. How to Tolerate Half Less One Byzantine

Nodes in Practical Distributed Systems, *23rd IEEE International Symposium on Reliable Distributed*, 2004

[17] A. Bessani, J. Sousa and E. Alchieri. State Machine Replication for the Masses with BFT-SMaRT, *Dependable Systems and Networks (DSN)*, 2013

[18] Y. Brun, G. Edwards, J. Young Bang, and N. Medvidovic. Smart Redundancy for Distributed Computation, *31st International Conference on Distributed Computing Systems*, 2011

[19] A. Clement, E. Wong, L. Alvisi, and M. Dahlin. Making Byzantine Fault Tolerant Systems Tolerate Byzantine Faults, *NSDI'09 : 6th USENIX Symposium on Networked Systems Design and Implementation*, 2009

[20] G. Santos Veronese, M. Correia, A. Neves Bessani, and L. Cheuk Lung. Spin One's Wheels ? Byzantine Fault Tolerance with a Spinning primary, *SRDS'09*, 2009

[21] A. Neves Bessani, M. Correira, P. sousa, N. Ferreira Neves, and P. Verissimo. *Intrusion Tolerance* : The "killer app" for BFT Protocols(?), https://www.net.t-labs.tu-berlin.de/~petr/BFTW3/abstracts/position-bft.pdf

[22]M. Vukolić. The Byzantine Empire in the Intercloud, *ACM SIGACT News*, 2010

[23] C. Pu, A. P. Black, C. Cowan, J. Walpole and C. Consel. A Specialization Toolkit to Increase the Diversity of Operating Systems, *ICMAS Workshop on Immunity-Based Systems*, 1996

[24] D. P. Anderson. BOINC : A system for Public-Resource Computing and Storage, *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, 2004