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

Efficient management of byzantine faults in clouds

Bournat Marjorie

Supervisor : Olivier Marin
Referent teacher : Julien Sopena

I. Introduction

Desktop Grids are used more and more to compute tasks or to stock data. A *Desktop Grid* is a distributed environment where many types of faults occur. Those faults range from crashes to complex faults like byzantine faults.

Byzantine faults correspond to arbitrary behavior of machines. They can result from software mistakes, hardware mistakes or attacks. This class of faults is the largest: it includes crashes and transient faults. The name “byzantine faults” comes from the analogy used by Lamport et al. [1] defining them.

Byzantine faults can break a system, and these kind of faults are more and more spread because of the complexity of applications (which increases the number of potential bugs permitting intrusions). Techniques have been developed to tolerate them: two stand out. On one hand are techniques that use hardware to deal with byzantine faults like [2] and on the other hand are techniques using software. Software solutions are divided in two families. The first family uses replication. When a client asks a scheduler for the execution of a query, the scheduler doesn't dispatch it to a single worker (machine that executes the query/task) but to several workers. In fact, if the scheduler dispatches the query to only one worker, and if this worker is byzantine, the scheduler will probably return (it isn't because a worker is byzantine that it will answer badly: it can try to fool the system with a few correct answers) an incorrect answer to the client. After the execution of the request, workers give their results to the scheduler which selects the result to send to the client. Very often the result sent to the client corresponds to the majority result received from the workers. In the second software method the scheduler forwards the client query to only one worker. However, it sends this worker *n* others queries. The scheduler knows the answers of those *n* other queries. When the scheduler receives all the answers (there are *n+1*), it can decide whether or not the result is correct. In fact, if the results for the *n* requests correspond to the *n* answers the scheduler knows, it assumes that the result for the client is correct too. Otherwise it needs to ask the *n+1* queries to any other worker.

We focus on replication techniques.

Algorithms that tolerate byzantine faults (named BFT for Byzantine Fault Tolerant) have multiple goals: they must guarantee properties of safety and liveness. Safety assures that the system behaves according to its specification and liveness ensures that the execution keeps on. Some algorithms guarantee the confidentiality of data. We don't care about this last characteristic, because our goal isn't to manipulate critical data but to ensure an efficient service that tolerates byzantine faults.

I.1. Principles of BFT

To tolerate byzantine faults with replication, the state of the art distinguishes deterministic methods and probabilistic methods.

<u>deterministic methods:</u>

With deterministic methods (like [4, 5, 6, 7, 8, 9, 11, 12, 14, 15, 16, 17, 19, 20]) the scheduler always gives 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. As

a byzantine fault can be a crash, if the scheduler waits for 3f+1 answers and there is at least one crash, the scheduler will wait indefinitely. So the scheduler waits only for the answers of 2f+1 workers upon the 3f+1 requested. But it is possible that among these 2f+1 answers, *f* are wrong. A non-byzantine worker can be slow, and so it is possible that the 2f+1 received answers correspond to the answers 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 to these methods. The choice of f is complex: 3f+1 can be big. If we choose a value f which is very large compared to the real number of byzantine faults in the system, we will waste resources. For instance, if we choose f=10, we replicate each task on 31 machines, yet if the real number of byzantine faults is 2, 7 replications would have been enough to give a correct answer to clients. The 24 other machines could have been used for others tasks, and so the throughput could have been better if f had been sized correctly. On the other hand, if we choose a value f smaller than the real number of byzantine faults in the system it is likely to provide false results 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 a maximum number of byzantine faults in the system. However the result returned to the client isn't always correct. Nevertheless it is possible to assess the reliability of the result.

To do that the scheduler (which in those algorithms is reliable, it doesn't crash and it isn't byzantine) maintains 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 assigns a default reputation value. Then during the execution this reputation will evolve: if the worker gives a correct answer (a correct answer corresponds for instance to an answer returned by the majority of the workers involved) the scheduler will increase the reputation of the worker, and if the worker gives a bad answer the primary will decrease its reputation.

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

This method has 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 required by the client, send the replicated task to the workers and send the final result to the client.

Notes: Probabilistic methods don't consider an agreement phase because they consider that all tasks are independent. It means that, unlike NFS, the system doesn't store global information that each task can modify.

In deterministic methods there isn't any reliable node. Here we consider a reliable node. The question that we could ask is: why not 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 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 into 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 common 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 a client won't wait a long time before obtaining the answer to queries. The algorithm has to ensure a good throughput. As well as elasticity. Elasticity corresponds to the adaptability of the algorithm in function of the workers present in the system (nodes can come and leave the system at any time). Each task has to be replicated on the fewest possible nodes. The algorithm mustn't make any assumption about the maximum number of faults in the system, and has to ensure good performance even if the number of faults increases. Finally the algorithm must be scalable, and so bear a great number of clients while 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 [4]	$O(f^2)$	5	3f+1	3f+1	No
Quorum de Aliph [5]	O(f)	2	0	3f+1	No
Quorum de HQ [8]	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 on the reliability the client wants for its answer	Yes
Q/U [12]	O(f)	2	0	4f+1 (but needs of 5f+1 machines)	No
Chain de Aliph [5]	O(f)	3f+2	1	3f+1	No
Zyzzyva [6]	O(f)	3	1	3f+1	No
Zlight de AZyzzyva [5]	O(f)	3	1	3f+1	No
UpRight[2] [7]	$O((u+r)^2)$	7	2u+r+1	u+r+1	No
BFTCloud [9]	O(f)	3	1	3f+1	yes
ByzID [11]	O(f)	3	1	2f+1	No
[14][3]	$O(f^2 + g*f)$	7 (9 if we use a	3f+1	2g+1	No

1 In the article the authors don't take into account the client. But if we considere the client we need 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 behaviors tolerated by the protocol ($u \geq r$)

		firewall)			
[15]	O(g*f)	5 (considering only one checkpoint otherwise 5+4*nbCheckpoints	3f+1	between g+1 and 2g+1	No
[16]	O(f)	3[4]	2f+1	2f+1	No
[17]	$O(f^2)$	5	3f+1	3f+1	Yes
Aardvark [19]	$O(f^2)$	5	3f+1	3f+1	No
Spinning [20]	$O(f^2)$	5	3f+1	3f+1	No

All these solutions incur a bottleneck when the number of requests increases. In fact all those algorithms rely on a single scheduler (also named *primary*), or on a coordinated group of machines, to wait the requests from clients, to order them on the workers, to wait for answers and to send the result back to the client.

Probabilistic solutions ([3, 10]) don't assume 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. We will focus on 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 and the assumptions we make. 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 as a comparison with 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:

We consider a distributed architecture similar to BOINC (Berkley Open Infrastructure for Network Computing) [13] with multiple reliable schedulers (also named *primaries*) and an infinite number of unreliable workers. The architectures for probabilistic algorithms of the state of the art are flat: there is only one reliable scheduler. So when a client sends a request, it contacts this *primary.* In the same way, when a worker wants 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.*

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.

In the case of our distributed algorithm, similarly to a centralized algorithm, clients and workers could know the identities of all the *primaries* of the system. This would permit a totally decentralized solution, thus 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* increases if the number of *workers* increases and vice versa. Therefore 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 are too many *primaries* compared to the number of *workers*, the *first-primary* discards some *primaries*. Clients that want to send a request now contact the *first-primary* which transfers the request to one of the *primaries* chosen in function of the allocation strategy (see section III.7). Similarly a worker which wants to join the system asks the *first-primary.* Our distributed solution is now a hierarchical centralized solution. Even if there is a centralized entity (the *first-primary*), we hope to preserve performance and scalability. In fact, this entity is only a dispatch and so by adopting a suitable data structure the *first-primary* can choose an appropriate *primary* rapidly. 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 described in figure 1 with only one *first-primary*, multiple *primaries*, and several *workers.*

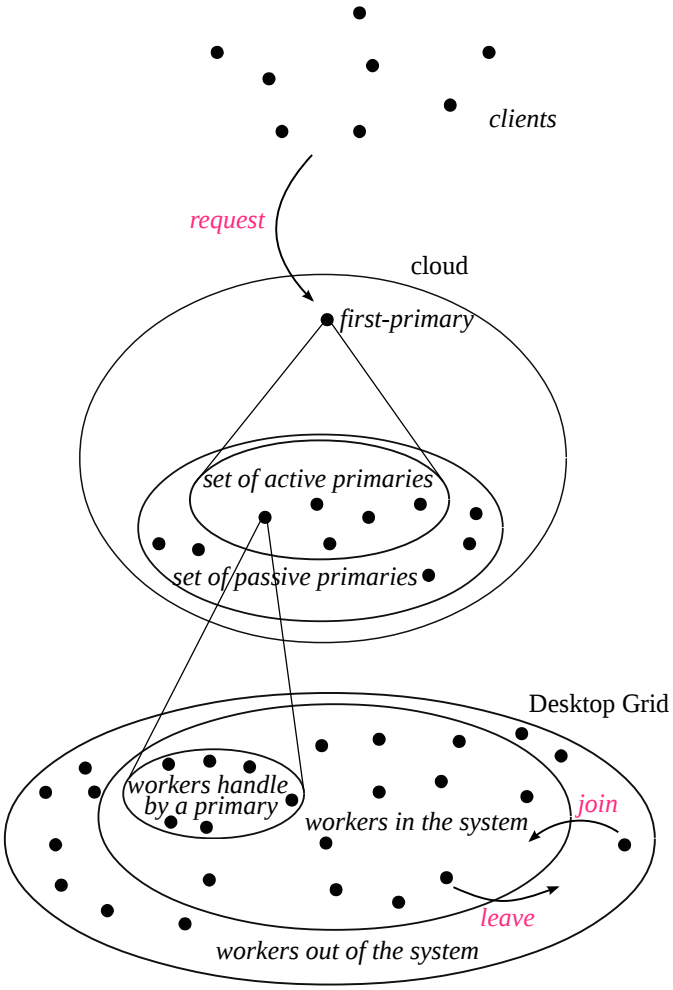

Figure 1 – general architecture of our solution

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

Our system must process a big quantity of requests from clients. Those requests correspond to big tasks 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 come at a price. Paying for a few reliable entities is reasonable, but the price can become prohibitive for a large quantity of machines. So we place our workers in an unreliable environment: the *Desktop Grid.* The environment is composed of a great quantity of workers. The nodes of this environment correspond to machines borrowed from people (like you) who have installed a program to authorize an external application to execute itself on their computer. When those people don't use their computer the external application can steal CPU cycles. When the owner of the machine needs to use his computer the external application can't execute itself. Some owners authorize the use of their CPU while they are using it, but the external application has to execute itself with a low priority, and so 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. Moreover a *Desktop Grid* application has to deal with byzantine faults.

II.2 Assumptions:

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

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

We suppose that all the faults are independent. This means any two faults have different causes. 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 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. 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 returns a 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 a bad answer to the client because of the collusion of byzantine nodes is weak. Moreover our system is based on a reputation system, which will limit requests to byzantine nodes (nodes that give bad answers multiple times will get a bad reputation).

We don't consider Sybil attacks. A Sybil attack is an attack where a node takes multiple identities. Thanks to all these identities, the node can be in superiority compared to the others nodes of the system. By feeding it false answers, 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 on multiple reliable *primaries.* Each *primary* handles a group of *workers* that execute tasks. The intersection between groups handled by *primaries* must be empty. Each *primary* schedules client requests on the workers it handles.

So we have to establish the way *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 subsets of *workers* beyond the workers handle by a *primary.*

We want the result to achieve a certain reliability defined by the client. First the *primary* will replicate on the minimal number of nodes necessary to reach the threshold. Thus, if all nodes answer the same response we can return the result to the client. If not, we have to replicate the task on additional *workers*. So we have to introduce strategies for additional replication.

Besides, *Desktop Grid workers* join (and leave) the system at any moment. We don't know in advance if a *worker* that joins the system will be reliable or not. Each *primary* gives a reputation to workers it handles in function of their actions all along the execution. So we have to study strategies permitting to update reputations.

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

Finally we describe the different ways for a client to choose the *primary* (via the *first-primary*) it wants to request.

All the different strategies will be compared. The goal is to know which is best in terms of latency, throughput, and minimal number of *workers* used to replicate a task. We will analyze these characteristics when there are no byzantine faults in the system and when there are byzantine faults. We will determine which strategy is better when there are faults in terms of result quality: which strategy permits to keep returning correct results when the number of faults increases. As scalability is important for us, we will investigate which solution produces 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 first describe the fields necessary for the *workers*, then those for the *primaries* and finally those for the *first-primary.*

Each *worker* has a structure W_i:

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

Each *primary* handles a set S of *workers*. For each *worker* it conserves the information below:

```
structure Si {
    Si.a     // answer returned by the node to the previous request
    Si.reputation     // reputation of the node
    Si.totR     // number of requests submitted to the node
    Si.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 Pi {
    Pi.A     // set of active workers under the responsibility of the primary
```

```
        Pi.P      // set of passive workers under the responsibility of the primary
        Pi.tmpSet     // set used during reset phases
        Pi.minR     // minimal reputation of workers that the primary handles
        Pi.maxR     // maximal reputation of workers that the primary handles
        Pi.churn     // number of the variation of workers leaving and joinig the system
        Pi.inPool      // boolean indicating if the primary is having nodes under his responsibility and
                          so receiving requests from client
        Pi.NotInF     // set of workers whose reputation require a change of primary
        Pi.able_to_send_division     // boolean used when executing the load balancing function
                                        (see notes of section III.2)
        Pi.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* keeps in memory the *primaries* participating to the application, and the reputation ranges 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 is not necessary. In fact, adding a node in the active set Fp_i.PA the *fisrt-primary* is just a new node allocation in the cloud, and removing a node from the inactive set Fp_i.PI is just a simple *first-primary* operation. 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 takes a node from Fp_i.PI and puts it in Fp_i.PA. Similarly when there are too many 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*

In this section we describe 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 another one in function of the reputations of *workers.*

Whatever the strategy used the joining of a node follows the protocol described in figure 2. The *worker* asks the *first-primary* to join the system. The *first-primary* computes an algorithm to know which *primary* it has to forward the request to.

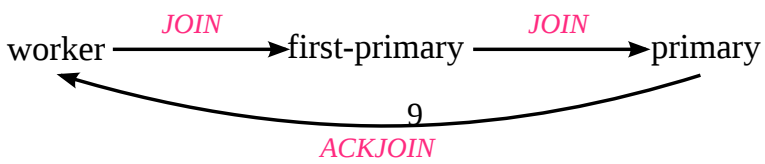

Figure 2 – addition of a new node

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

III.2.a. Random strategy

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

Uniform distribution allows the assumption that nodes also leave the system uniformly. So the two previously introduced properties remain true even when nodes leave. However, we can describe a reset strategy activated after the arrival or the departure of a lot of nodes. The reset ensures that 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* notices that there are too many 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 U {W_i}
P_i.churn++
if abs(P_i.churn) > x **then**
call *reset*
end if

Similarly a node that leaves the system informs its *primary*. Sometimes, some *worker*s crash and thus leave the system without being able to inform their *primary. Primaries* use timers to detect crashes.

---

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 maintains well-balanced groups of *workers* in terms of number and of reputation. When executing this function *primaries* send messages to each other to indicate the number of nodes they handle, and to determine the size T of the smallest group of *workers*. Each *primary* keeps T *workers* under its responsibility and asks 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 U {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 rarely be 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 reputations. For instance a *primary* can have under its responsibility nodes whose reputation is between 98 and 100 whereas another *primary* handles nodes whose reputation is between 96 and 98... To balance groups sizes, multiple *primaries* can handle the same range of reputations.

When a *worker* joins the system, there are several strategies. We could decide not to trust the new *worker* (it is the pessimistic strategy: we consider that all 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 results to clients. When we consider

nodes of high reputation the system requests 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. Since we use the optimistic strategy, the *first-primary* assigns them to the same *primary* which handles nodes of high reputation. If the *primary* requests 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. Hence if a byzantine node is in the system, it has to give lots of good answers to acquire 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 faster.

---

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 following technique is employed:

---

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 \ {W_i}
else
P_i.A = P_i.A \ {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 reassign nodes during the execution in a dynamic way. Indeed, each *primary* handles a given range of reputations, and nodes reputations evolve during execution. After the update of the reputations of nodes it handles (see section III.5 talking about the update of reputations) each *primary* checks that its nodes still have a reputation included in the range of reputation it handles. If it isn't the case 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 $w_k \in [P_j.minR, P_j.maxR[$

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 nodes have to change their *primary* regularly, this method is expensive. In this case, we could wait a certain time, or until a fixed number of nodes request a change of their *primary* to enfirce these modifications. This would reduce the number of messages. 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 store the set of nodes whose reputation doesn't correspond anymore to the range of reputations handled 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 ensure that a node which wants to join or leave the system will be able to do so eventually.

- It is possible that a *primary* hasn't enough nodes under its responsibility to be able to answer the request of clients. Or conversely, it is possible that a *primary* handles too many nodes (and so compute strategies in a slowly way). So we need a *load balancing* mechanism to maintain groups reasonable number of nodes to execute requests in a fast way.
  When a *primary* won't have enough nodes to answer clients, it will call the *load balancing* function. This function deactivates the *primary* putting its inPool boolean to false. The *workers* under the responsibility of this deactivated *primary* will be redistributed on the active *primaries*.
  Similarly if a *primary* is in charge of too many *workers* the system redistributes the overload among the other *primaries.*

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

Below in black is the pseudo-code of the random *load-balancing* function. Blue/green writings correspond 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

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, *primary* P_i in the pseudo-code won't consider the worker W_j under its responsibility after the execution of the *load balancing* function. P_k will consider the worker under its charge, and after if the *worker* W_j decides not to answer P_k, P_k will just consider it has crashed and will remove 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* performs group formation. This means that each *primary* forms subsets of *workers* beyond the nodes it handles. Several formation strategies are possible: *fixed-fit, first-fit, tight-fit, random-fit* and the method of Arantes et al. [3]. The *primary* submits the same task to all *workers* belonging to the same subset.

The four first methods correspond to strategies used in the article written by Sonnek et al. [10] (which is a centralized probabilistic algorithm). The last method corresponds to the method presented in the article of Arantes et al. [3]. Our hierarchical solution is implemented above these 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 group is fixed statically (by an administrator for instance). The *primary* randomly selects *nb workers* to form a subset.

Note:

When *workers* are assigned to *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 by Sonnek et al. the *first-fit, tight-fit* and *random-fit* strategy form only odd-numbered 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.

In the *first-fit* strategy *workers* are ordered in descending order of reputations. The *primary* regroups *workers* (following the ordered list) until achieving threshold $\lambda target$, or until the group contains the maximum authorized number of nodes. 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* authorized in each group too.

This algorithm corresponds to the one described 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 a 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* to *primaries* considers the reputations, the *first-fit* strategy forms group in a random way (see section III.3.d for the description of the random formation group). In fact since a *primary* handles *workers* with similar reputations, the preliminary sort of nodes is unecessary.

III.3.c Tight-fit

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

This strategy is surely more expensive than the *first-fit* strategy, but it provides more groups that are more balanced in terms of reliability.

The pseudo-code of this strategy is the following:

---

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

We want to form odd-numbered groups. When using "binary search to identify the smallest set G_r of *n workers* from G" we search for two workers. These two workers are side by side in the descending order of reputations and their reputations are above 50.

The following example helps explain the principle:
Assuming we have 14 workers with the following reputations:

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;" and "$G \leftarrow G \setminus \{W_r\}$" forces 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 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 nodes are added to groups in a random way and considering their reputations. The *primary* adds nodes in a group until it achieves threshold $\lambda target$ (and the minimal number of nodes in a group), or until the group contains the maximum authorized number of nodes.

This strategy is different from the *first-fit* strategy because *workers* are added in a random way to a group.

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

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 selected nodes are byzantine. P_C corresponds to the probability that all the nodes in the group are correct. *th* is the threshold we want to achieve.

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

When using the Arantes and al. method we can form either odd- or even-numbered groups.

III.4. Additional replication strategies

The goal of a probabilistic method is to request the minimum number of machines necessary 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 nodes don't return the same answer, this means that the threshold of reliability for the answer isn't achieved. So we need to replicate the task on additional *workers* until achieving this threshold. Sonnek et al. want a majority answer to return the result. If the majority isn't achieved they just throw an exception to the client.

So in this section we will detail the replication strategy for Arantes et al. We will also detail the *progressive* and *iterative redundancy*, that are described in the article [18] 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. [3]. 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.

The pseudo-code of this algorithm is:

---

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

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 return the most frequent value (this means that we obtained a subset of *workers* such that the formula2 is less than (1 - λ*target*)) then we can returned the result to the client. If not we have to compute again the additional replication function.

III.4.b. Progressive redundancy

In the general way, to tolerate the byzantine faults we replicate a task over k *workers* and then we return the majority answer ($\lfloor \frac{k}{2} \rfloor + 1$). *Progressive redundancy* is an optimistic method used to reduce the number of *workers* requested. When we use this technique we first request only $\lfloor \frac{k}{2} \rfloor + 1$ *workers.* If all nodes give the same answer then a consensus is reached 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 gives another answer, we have to replicate the task on another 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 there is a majority answer returned by 4 nodes, it is this answer that we will return 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 8 nodes have already been requested, and if 4 returned a result and 4 others another result, we have to replicate again to decide what the final result is.

III.4.c. Iterative redundancy

In this strategy the primary distributes tasks to nodes in order to achieve a certain threshold of reliability. However, the authors that introduced *iterative redundancy* [18] show 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, there 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 nodes 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 *d* between the correct and incorrect results to achieve the reliability threshold wanted.

So, the replication will occur repeatedly until *x* machines give the bad result and *x+d* give the good.

As we indicated this technique considers that all nodes have the same reputation. However, we could analyze the behavior of this technique in our system despite the evolution of reputations.

III.5 reputations update strategies

The *primary* must update the reputations of *workers.*

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

We need an algorithm 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 answers returned by the requested *workers.* Each answer will be weighted by the number of *workers* that have returned it. 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 fills this structure.

If our hierarchical solution is based on Sonnek et al. algorithm we will return *A* to the client if *nb*=1 and $x \geq \lfloor \frac{tot}{2} \rfloor + 1$. In this case *workers* that returned *A* have their reputations increased and *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 verified then the reputation of all the *workers* requested for the task will see their reputations decreased.

With 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 with the biggest value for formula2. In this case if *nb*=1 and (*x = tot* or $x < (1 - \lambda target)$) then we return *A* to the client, increase the reputations of *workers* that returned *A,* and decrease the reputations other *workers.* If the conditions aren't checked we have to replicate until the condition verifies to update the reputations of the *workers.*

Below we put the pseudo-code of the algorithm. We write it as if we were using Sonnek et al. but it is quite the same algorithm used by Arantes et al. (just change x with 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 strategies that we will use to update reputations in our solution: *symmetrical*, *asymmetrical* and the update presented in the article written by Sonnek et al.

III.5.a. Symmetrical

The *symmetrical* strategy increases or decreases the reputations by 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 Si.a == v then
                Si.reputation =  Si.reputation + Si.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 gain the trust of the system slower than it will lose it. In case of a bad answer, the application will penalize the node strongly.

III.5.c. Sonnek et al. [10] update of reputations method

The strategy used by Sonnek et al. consists in counting the number of assumed correct answers provided by a node, and the number of requests that have been submitted to it. The reputation of the node is then computed with the formula: S_i.reputation = (S_i.totC + 1) / (S_i.totR + 2).

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

III.6. Complements

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

When the reputation of nodes is too low, latency increases. In fact, if we use a group formation strategy based on the reputations of nodes, we request more nodes. We have to wait for the answers of those nodes or to detect their failures, so this is slower than if we request less nodes.

Moreover since nodes have bad reputations, it is probable that we have to do some additional replication to obtain the answer. So the latency increases. To avoid this we could use a blacklist strategy. This means that we can remove nodes which have their reputations below a certain threshold. For instance we could remove all nodes that have reputations below 50.

Moreover, some byzantine nodes temporarily give good answers in order to have their reputations increased. Once they have a good reputation, they can send bad answers to the system without necessarily getting caught by the system. To avoid this situation we could regularly reset the reputation of all nodes of the system to an initial value. If the assignment of *workers* to *primaries* depends on reputations, nodes have to change their *primary* when the reset of reputation takes place.

We will make experimental comparisons between systems that have those mechanisms of blacklist and reset of reputations, and systems without these techniques.

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 also request a *primary* depending on the cost it is ready to pay.

III.7.a. Random

This strategy permits 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 suitable when the assignment of *workers* to *primaries* is done

considering their reputations.

Indeed, when *primaries* are divided in ranges of reputation, some *primaries* are responsible for *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 wants 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 remain a dispatch the client has to specify 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 which *primary* it has to forward the request to.

We focus on the method where the *first-primary* only dispatches:

---

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. Only one *primary* handles all requests.

These algorithms don't have to implement the assignment of *workers* to *primaries* nor the *primaries* interrogation (by clients) strategies. In fact, all *workers* are handled by only one *primary* and clients request this single node.

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

We describe the strategies used by the two algorithms with which we compare our solution.

IV.1. Strategies used by Sonnek et al.

The centralized algorithm of Sonnek et al. assumes that there is no collusion. 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 *fixed-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 reputation update strategy is presented in the section III.5.c., where the value is initialized to ½ and then is modified depending on the ratio of assumed correct answers.

The article doesn't consider additional replication. In Sonnek et al. if an absolute majority isn't obtained we send “fail” to the client, indicating that if it wants an answer it has to send its request to the *primary* again.

Sonnek et al. form groups with respect to node reputations (unless the group formation strategy is *fixed-fit*). A majority of nodes has to agree on the same result to return it. Here we could think that there is no sense in creating groups according to reputation and then wait for a majority. Indeed, imagine we form a group of 5 nodes, 3 of them with a reputation of 0.3 return a result x, the two other nodes give the same result *y* with a reputation of 0.65. Intuitively we want to send *y* to the client. However when processing Sonnek et al. the result sent is *x*. They do 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 byzantine nodes are authorized to collude.

The group formation strategy used in Arantes et al. is described in section III.3.e.

In this algorithm the *primary* isn't authorized to send “fail” to the client, hence the additional replication mechanisms presented in section III.4.a.

Several reputation strategies are presented in Arantes et al.: the symmetrical and the asymmetrical one, and also the one used in BOINC.

In BOINC [13] the reputations of nodes are computed from the error rate of 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 reputations. 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 either an emulator or 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 [21, 22]. 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. A task in SimGrid is sometimes fictive. It just corresponds to a the time the task will take, the size of the data of the task. In real application a task corresponds to a request, to computation to execute, here, none of these information are present. 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 (which are respectively in the "src" directory and in the "include" directoy):

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 [23]. 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).

We have a directory named "platforms" that contains multiple xml platforms and deployment files.

platform.xml: all the files describing platforms contains information to create a network with direct links only. All links are used in a symmetrical way. This means that when a link *l* permits to join a node *n1* to a node *n2*, when *n2* wants to send a data to *n1*, the data are send through *l*.

For instance figure 3 shown the network in the case we are simulate our hierarchical solution with one *first-primary,* two *primaries,* and 4 *workers.* The *first-primary* is linked directly with one

link with each *primaries*. Similarly a *primary* is linked thanks to a link with each *workers* of the system.

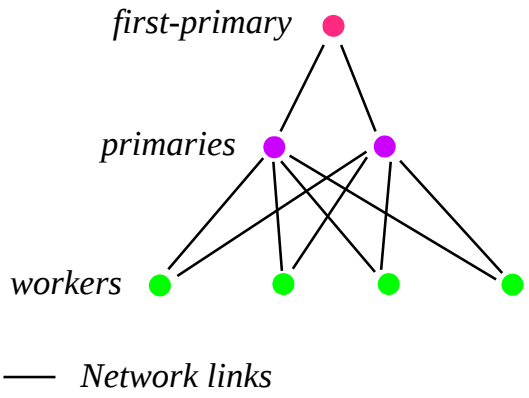

Figure 3 – network of the hierarchical solution

Even if nodes can be absent of the system (because of a crash), and even if a *primary* don't handle all the *workers* of the system, or if a *primary* start its execution in the inactive groups we define links between all *primaries* and all the *workers*, and create all the processes active one and inactive one. In fact, as we are in a simulation we decide to create all the process in advance, they will just been integrated in the system during the execution.

The difference between files describing platform in the “platfoms” directory lies in the presence or absence of a *first-primary*, the number of *primaries*, the number of *workers*, and the number of *clients* (corresponding to the charge of the system).

deployment.xml: the informations contained in the deployment file for the centralized and hierarchical solutions are different.

For the centralized solutions the *workers* and *clients* have as argument the identity of the *primary*, whereas for the hierarchical solution they have the identity of the *first-primary*. For the laster solution, the *first-primary* have as argument of the list of the *primaries'* identities (whether the *primaries* are active or inactive).

For all the different solutions, all the processes (*first-primary*, *primaries*, *workers*, clients) have as argument their own identity (this serve to know on which mailbox receive messages).

Also for all the solution the *workers* have as argument a number indicating the real reliability *r* they have. When a *worker* receive a task to accomplish, if it hasn't crash before receiving the task or if it doesn't crash during the fictive execution of the task, it has to give a result to its *primary*. To do that it will randomly chose a value *v* between 0 and 100 included. If the $v > r$ then the *worker* has to give a bad answer to the *primary*, if not it returns a good answer.

The value returned to symbolize a good answer is the value 0, and for a bad it depends. If we authorized collusion a bad answer corresponds to the value 1. If the collusion isn't authorized, the *worker* chose a value between 1 and a value big enough in order the probability that two or more *workers* in the same groups give the same answer is small.

V.2.2. details about the implements in the simulator

When using simulation with SimGrid, the are only few ways to increment the simulation time. To increment it, there are 3 main functions (and those which are similar to them) that will be

useful:

msg_error_t MSG_process_sleep(double nb_sec): this function permits to a node to sleep during the time (in seconds) indicating in parameter. If all the nodes are sleep then the simulation time will be increased until one of the process wakes up.

msg_error_t MSG_task_send (msg_task_t task, const char * alias): this function permits to send a task (previously created). When the destination received the message the simulation time is increased until to the reception time which is computed considering the size of the task, the emitter and the destination of the task, so considering the route and the latency/bandwidth of each links contained in the route.

msg_error_t MSG_task_execute (msg_task_t task): This function simulate the execute of a task. No computation are executed from the process which run this function. The process go out the function increasing the simulation time until the time is must have finish considering the flop to execute in the task and considering the power of the CPU (in flop/s) on which the task is executing.

This implies that when we are executing local algorithm on a process, for instance we are browsing a table, the simulation time isn't incremented.

As we want to compare different strategies (section III), and see what is the strategy that offer the better latency, the better throughput, we have to know the time taken in the local algorithm. So we have to calculate the complexities of each algorithm in flop, create a task with this quantity of flop and then execute it thanks to the function MSG_task_execute after each algorithm important in our comparison.

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] L. Arantes, O. Marin, P. Sens and R. Friedman. Probabilistic Byzantine Tolerance for Cloud Computing, *34th International Symposium on Reliable and Distributed Systems*, 2015

[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] D.P. Anderson. BOINC : A System for Public-Resource Computing and Storage, 5th IEEE/ACM International Workshop on Grid Computing , 2004

[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] H. Casanova, A. Legrand, M. Quison. SimGrid : a Generic Framework for Large-Scale Distributed Experiments, Tenth International Conference on Computer Modeling and Simulation, 2008

[22] http://simgrid.gforge.inria.fr/

[23] http://fta.scem.uws.edu.au/