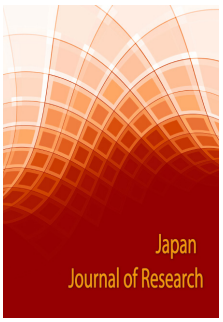
Japan Journal of Research

Review Article

Model for Anticipating Frank Failures in Computing Grids

Ramadane Adamou Yougouda, Vivient Corneille Kamla, Laurent Bitjoka

ENSAI, University of Ngaoundere, Ngaoundere, Cameroon



Correspondence

Ramadane Adamou Yougouda

ENSAI, University of Ngaoundere, Ngaoundere, Cameroon

- Received Date: 17 Sep 2024
- Accepted Date: 30 Sep 2024
- Publication Date: 01 Oct 2024

Keywords

computational grids, fault tolerance, hard failures, node, PDEVS, replication, backhaul, modeling, simulation, DEVSJAVA.

Abstract

Computing grids are infrastructures that provide almost infinite computing capacities, they are now used in all fields, from the study of pandemics to the monitoring of rocket trajectories and the study of meteorological and climatic phenomena. They have a distributed and heterogeneous architecture that gives them unlimited computing performance. They are made up of several computing nodes that are subject to failures like frank failures. A frank failure in a computing grid is an abnormal and unexpected interruption of a node. Many frank failure tolerance protocols have been proposed in the literature but none of these protocols integrate the anticipation of frank failures. The objective of this article is to propose a model based on the PDEVS formalism of frank failure tolerance in a computing grid that allows the anticipation of frank failures. The proposed model relies on the temperature variation of electronic components and on the state of the hard disk through the values provided by SMART data to predict a probable frank failure of a node. The results of the simulations on the different scenarios that we have carried out show that our results provide better performances than those proposed in the literature when the number of nodes to tolerate is greater than 200.

Introduction

An element is said to be faulty when it does not behave in accordance with its specification. A failure is often called a "breakdown". There are several types of failures, grouped into three families: frank failures which are failures that definitively stop the activity of the node that suffers the failure; Failures by omission are failures that temporarily stop the activity of the node that suffers the failure; and Byzantine failures are failures that do not stop the activity of the node that suffers it but behaves in an unexpected way [1]. In this article we will focus on frank failures. When a frank failure occurs in a node of a computing grid, the task that this node was executing is lost. Several fault tolerance methods have been proposed in the literature, but the main methods that have been implemented are: the replication method which consists of having several nodes that each process a copy of the job and in case of failure of the main node, one of the nodes that processes a copy of the job replaces the failed main node and the back-up method consists of saving in a storage medium, at a defined period, a copy of the job that is processed. Once the main node fails, the last backup is reassigned to a node and the execution of the job resumes [2]. Several fault tolerance protocols implementing these two methods have been proposed. The CoCheck protocol is the very first implemented protocol, based on the recovery method. It was developed by Stellner in 1996 [3]. Several other fault tolerance protocols based on the recovery method have been proposed over time: MPICH-V presented by Bosilca in 2002 [4], Kaapi-TIC presented by Jafar in 2005 [5], Open MPI presented by Hursey in 2007 [6]. Although the protocols based on the recovery method tolerate frank failures well, the resumption of the job execution starts again at the last backup and it even happens that another failure occurs during the copy of this backup to the node that must continue its execution. Other protocols based on the replication method have been proposed [7]. Replication methods are methods that rely on the principle of consensus between the different nodes to determine the best replica to use to replace the failed node [8] [9]. The very first protocol based on the replication method is the PBFT proposed by Castro in 1999. [10] Several other protocols based on the replication method have been proposed, we can present: The Q/U protocol was presented in 2005 by Abd-El-Malek [11], Kotla presents Zyzzyva in 2009. [12]Although several fault tolerance protocols based on the replication method have been proposed for frank failures [13], they require many nodes to save the copies and high performance to execute copies simultaneously. Regardless of the method implemented by the different fault tolerance protocols, they wait for the failure to occur during the execution of the tasks before being able to apply the tolerance

Citation: Yougouda RA, Kamla VC, Bitjoka L. Model for Anticipating Frank Failures in Computing Grids. Japan J Res. 2024;5(9):065

method implemented by the protocol. The question for us in this article is therefore whether it is possible to propose a model for anticipating frank failures in a computing grid?

The objective of this paper is therefore to propose a fault tolerance model that anticipates the frank failures that can occur during the execution of tasks in a computing grid.

The plan of this article is as follows: we start by presenting a modeling and simulation of frank failures in a computing grid. Then we propose models that allow the anticipation of frank failures that occur in computing grids during the execution of tasks. And finally we make a comparative study of the different fault tolerance methods.

Calculation grid and modeling of frank failures

For the modeling and simulation of our computing grid, we will use 100 computers of different characteristics representing our 100 nodes which will be simulated using the PDEVS formalism. In this part we present the computing grid and the formalism used for modeling and simulation of frank failure anticipations.

Presentation of a calculation grid

A computing grid is defined by Buyya, as a type of parallel and distributed system that allows the sharing, selection, and aggregation of autonomous resources geographically distributed dynamically. Each of these resources has its own availability, capacity, performance, cost, and users, with their own quality of service constraints [14]. The computing grid is considered as the technology of the future [15]. It can process in a few weeks what an ordinary supercomputer would take years to process. For example, we can cite here as an example the Help Care Muscular Distrophy (HCMD) project. The aim of this project was to study protein-protein interactions for more than 4000 human proteins [16]. To complete this study with a computer equipped with a processor clocked at 2 gigahertz, 1,489 years would have been necessary. However, this project only lasted 06 months with the computing power of the World Community Grid (WCG). This grid pooled the computing power of the computers of several volunteers [17]. The powers of computing grids have been used in other projects requiring very high computing power such as: For the analysis and prediction of climatic phenomena [18]. For the visualization, analysis, storage and sharing of images [19]. Although today computing grids are used in the largest projects that require high computing power. Its architecture is due to many frank failures. In the following part, we will present the signs that announce a frank failure.

Causes of a clear failure

Frank failure can happen at any time in a grid and cause us to lose data. This failure can often have precursor signs or warning signs of frank failures. In this part, we will see the node elements to monitor to anticipate frank failures in computing grids.

The fan

A computer fan is a component whose purpose is to maintain the temperature of sensitive computer components at an optimal level. When components such as the processor or graphics card reach excessive temperatures, this can lead to failures. The fan rotation speed varies automatically depending on the thermal conditions and the configuration. The minimum and maximum duty cycle parameters as well as the primary and secondary temperature input parameters are used to specify the configuration. It is therefore a question of proposing a model to ensure that the minimum rotation speed of the fan is never lower than the minimum rotation speed recommended by the manufacturer and that the maximum rotation speed of the fan is never higher than the maximum rotation speed recommended by the manufacturer.

SMART data

SMART data, known as self-monitoring, analysis and reporting technology are considered as one of the first signs of hard drive failure. Once this is enabled, it will usually alert you that your hard drive is having problems when you try to boot it. However, the sad part about this is that most brands do not have this option to be automatically enabled. All machines have this option since 2004 but SMART data is not automatically enabled on all machines. There are tools that aim to predict hard drive failures by reading the data recorded by the operating system. If you have a working hard drive and want to take a look at its SMART disk, you need to open your web browser. Each of the data is measured and compared to its worst value or threshold value. Now this worst value can be adapted according to the context and conditions of use of the machine [20].

Description of a calculation grid

Ian Foster and Carl Kesselman were the first to use the word "grid" in their book "The Grid: Blueprint for a New Computing Infrastructure" published in 2003. [21]They compare a computing grid to an electrical grid in terms of resource availability. In an electrical grid, you simply plug in an outlet and electricity flows. A computing grid is defined as a set of shared, distributed, heterogeneous, delocalized, and autonomous hardware and software resources in which large-scale scientific problems can be solved. The lifecycle of a task in a computing grid begins when the user obtains a certificate from a certificate authority trusted by the grid. The user then registers with a virtual grid organization and obtains their user certificate. The user can then log in to the platform from a particular machine that serves as a user interface from which they can launch a task. This task is submitted to the work management system (WMS) from the user interface [22]. The WMS searches for available compute elements (CEs) or nodes to process the task. Once the processing of the task parts is complete, the WMS rebuilds the processed task. If a hard failure occurs during the execution of the task, for example if tasks are sent to a node that has shut down, the task that was being executed is lost.

The PDEVS formalism

Formal specifications of the model

The PDEVS (Parallel Discrete EVent system Specification) formalism was introduced in the early 70s by Mr. Zeigler [23] who aimed to provide a solid mathematical foundation for the modeling and simulation (M&S) of discrete event systems. A discrete event system is a system that can be described from a set of states and transition rules between these states. The PDEVS formalism allows the representation of a system as a model or a set of models having states and transitions. In addition, it provides the possibility to define the structure of a system in a distinct manner. The PDEVS formalism presents the formal definitions of atomic and coupled models. We present in the rest of this paper the formal specifications of these different models.

Atomic models

An atomic model MA In the PDEVS formalism is specified by the definition of eight entities

$$MA = \langle X, Y, S, \delta_{\text{int}}, \delta_{ext}, \delta_{con}, \lambda, ta \rangle$$

With :

- The set of inputs : X = {(p,v) | p ∈ Pe , v ∈ Vx} Where Pe and Vx are two finite sets representing the set of input ports and the values carried by the events received as input;
- The set of outputs : Y = {(p,v) | p ∈ Ps , v ∈ Vy} where Ps and Vy are two finite sets representing the set of output ports and the values carried by the events generated at the output;
- The set of states: S = {s i | i ∈R + } ;
- The internal transition function δ int (S) →S ;
- The external transition function δ ext (Q x X) →S , where, *X* b is the set of input bags belonging to X, Q is the set of total states, Q = {(s, e) | s ∈S, 0 ≤ e ≤ ta(s)}, e is the time elapsed since the last transition;
- The confluent transition function δ con (S x X) →S ;
- The output function : λ (S) → Y ;
- The time advance function : ta(S) → R_0^+∪ ∞.
- Coupled models

In the classical PDEVS formalism with port, a coupled MC model is specified by the definition of seven entities:

$$MC = \langle X, Y, D, EOC, IC, EIC, select \rangle$$

With :

- The set of inputs : X = {(p,v) | p ∈ Pe , v ∈ Vx} Where Pe and Vx are two finite sets representing the set of input ports and the values carried by the events received as input;
- The set of outputs : Y = {(p,v) | p ∈ Ps , v ∈ Vy} where Ps and Vy are two finite sets representing the set of output ports and the values carried by the events generated at the output;
- The set of atomic or coupled models : D = {m i | i ∈ R + } ;
- The set of external output couplings: EOC = {((m i , p sj), (MC, p sk)) | i, j, s, k ∈ R + } ;
- The set of internal couplings : IC = {((m i , p sj), (m k , p el)) | i, j, s, k ∈ R + } ;
- The set of external input couplings: EIC= {((MC, p ei) ,(m j , p ek)) | e, i, j, k ∈ R + } ;
- The selection function to adjust the activation conflict of the select models (D) → m i.

Graphic Model

The PDEVS formalism not only describes the model with the formal specifications but also by the graphical model proposed in [24]and represented in Figure 1 .

$$MA = \langle X, Y, S, \delta_{\text{int}}, \delta_{ext}, \delta_{con}, \lambda, ta \rangle$$

Grid Computing Simulation

For this simulation, we used the devs-suite simulator version 4.0.0 developed by Arizona Center for Integrative Modeling and Simulation (ACIMS) using DEVSJAVA as language [25]. To simulate the model below, the characteristics of 100 nodes were used and summarized in the table below. And for the code we used Eclipse IDE for Enterprise Java Developers.

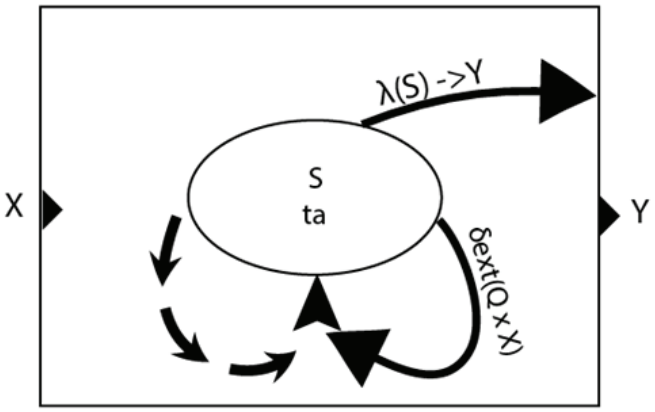


Figure 1. Graphical Model with PDEVS

With:

- X = Set of entries;
- Y = Set of outputs;
- δ ext (QxX) = External transition function;
- Internal transition function;
- λ (S)->Y = External transition function;
- Confluence transition function;
- S ta — Set of sequential states of the model.

Table 1. *Characteristics of the nodes used for modeling*

Node	CPU	Cores	Tcase	RAM size
x 10	2 x Intel Xeon Gold 6130	16 cores/ CPU	87°C	192 GiB
x 15	2 x Intel Core i5-3320M	4 cores/ CPU	73°C	12GiB
x 15	5 x Intel Core i5-3320M	4 cores/ CPU	73°C	12GiB
x 10	2 x POWER8NVL 1.0	10 cores/ CPU	74°C	128 GiB
x 10	2 x Intel Xeon Gold 5218	16 cores/ CPU	87°C	1.5 TiB
x 3	10 x Intel Xeon Gold 6130	16 cores/ CPU	87°C	768 GiB
x 20	2 x Intel Xeon Gold 6130	16 cores/ CPU	87°C	768 GiB
x 10	5 x Intel Xeon E5-2630 v4	10 cores/ CPU	74°C	256 GiB
x 2	2 x AMD EPYC 7301	16 cores/ CPU	65°C	128 GiB
x 20	16 x AMD EPYC 7301	16 cores/ CPU	65°C	128 GiB
x 45	2 x Intel Xeon E5-2680 v4	14 cores/ CPU	86°C	768 GiB
x 10	4 x Intel Xeon Gold 6126	12 cores/ CPU	86°C	192 GiB
x 10	2 x Intel Xeon E5-2630L	6 cores/ CPU	69.8°C	32 GiB
x 20	9 x Intel Xeon E5-2630L	6 cores/ CPU	69.8°C	32 GiB

x 10	2 x Intel Xeon E5-2698 v4	20 cores/ CPU	90°C	512 GiB
x 50	16 x Intel Xeon E5-2620	6 cores/ CPU	77.4°C	32 GiB
x 10	10 x AMD EPYC 7642	48 cores/ CPU	66°C	512 GiB
x 40	AMD EPYC 7642	48 cores/ CPU	66°C	512 GiB
x 10	2 x Intel Xeon E5-2620 v4	8 cores/ CPU	74°C	64 GiB
x 20	2 x Intel Xeon E5-2630	6 cores/ CPU	77.4°C	32 GiB
x 50	7 x ¬ThunderX2 99xx	32 cores/ CPU	74°C	256 GiB
x 10	10 x ThunderX2 99xx	32 cores/ CPU	74°C	256 GiB
x 10	2 x AMD Opteron 250	1 core/ CPU	65°C	2 GiB
x 10	2 x Intel Xeon E5-2630	6 cores/ CPU	77.4°C	32 GiB
x 10	2 x Intel Xeon Silver 4110	8 cores/ CPU	77°C	128 GiB
x 10	8 x Intel Xeon E5-2630 v3	8 cores/ CPU	72.1°C	128 GiB
x 10	2 x Intel Xeon E5-2620 v3	6 cores/ CPU	72.6°C	64 GiB
x 10	2 x Intel Xeon E5-2650	8 cores/ CPU	77.4°C	256 GiB
x 100	8 x Intel Xeon Gold 5218R	20 cores/ CPU	87°C	96 GiB
x 10	2 x Intel Xeon E5-2650	8 cores/ CPU	77.4°C	64 GiB
x 8	Intel Xeon E5-2650 v4	12 cores/ CPU	80°C	128 GiB
x 12	4 x Intel Xeon E5-2650 v4	12 cores/ CPU	80°C	128 GiB
x 10	2 x Intel Xeon E5-2603 v3	6 cores/ CPU	72.8°C	64 GiB
x 20	16 x Intel Xeon E5-2630 v3	8 cores/ CPU	72.1°C	128 GiB
x 60	Intel Xeon E5-2630 v3	8 cores/ CPU	72.1°C	128 GiB
x 10	10 x Intel Xeon Gold 5220	18 cores/ CPU	87°C	96 GiB
x 40	2 x AMD EPYC 7452	32 cores/ CPU	65°C	128 GiB
x 50	AMD EPYC 7351	16 cores/ CPU	66°C	128 GiB
x 10	2 x Intel Xeon Gold 6130	16 cores/ CPU	87°C	192 GiB
x 10	2 x Intel Xeon E5-2660	8 cores/ CPU	73°C	64 GiB
x 70	2 x Intel Xeon E5-2630L v4	10 cores/ CPU	62°C	128 GiB
x 10	2 x Intel Xeon E5-2660 v2	10 cores/ CPU	75°C	128 GiB
x 10	3 x Intel Xeon X5570	4 cores/ CPU	75°C	24 GiB
x 10	12 x AMD Opteron 6164 HE	12 cores/ CPU	65°C	48 GiB
x 10	2 x Intel Xeon E5-2630 v3	8 cores/ CPU	72.1°C	128 GiB
x 80	Intel Xeon E5-2630 v3	8 cores/ CPU	72.1°C	128 GiB
x 10	10 x Intel Xeon X5670	6 cores/ CPU	81.3°C	96 GiB

For this modeling, we consider that the distribution of tasks in the calculation grid is perfectly balanced, that is to say:

$$Time = \frac{NbFlops_{totat}\left(\text{nœud1}\right)}{Flops/s\left(\text{nœud1}\right)} = \frac{NbFlops_{totat}\left(\text{nœud } N\right)}{Flops/s\left(\text{nœud } N\right)}$$

With: NbFlops the number of operations performed on all processors of a node. Flops/s the computing power of a node and Time the execution time of each node of the grid. The tasks are submitted to the system in an identical time interval.

Results and discussion

In this part we will start by presenting a calculation grid that works correctly without receiving a failure, then we will present a calculation grid that suffers a clear failure.

Modeling a correct grid

We first present the behavior of a correct node during task execution.

Model of a correct knot

The atomic model when modeling a computing grid is a model of a node in the grid. A node in a grid is a computing element of the grid. Nodes can be supercomputers, servers, PCs, clusters and even other grids. All components are networked using of equipment such as routers, cables, switches. The model of a node that we present here is a correct node model, that is, a model that cannot receive faults during the execution of tasks.

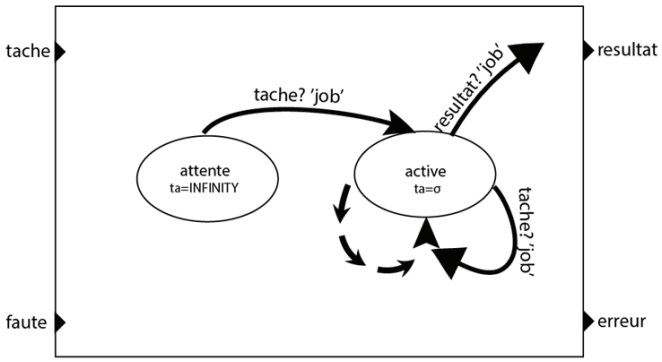


Figure 2. *Model of a correct knot.*

This model consists of two input ports and two output ports. The system is in an initial state "waiting". Once a job is sent by the task port, the node goes into an "active" state. At the end of each job processing, the result is returned by the output port "result". If another job is sent by the "task" port the system remains in an "active" state. The formal description of the model of a node is as follows.

$noeud = (X_{Noeud}, Y_{Noeud}, S, \delta_{ext}, \delta_{\text{int}}, \delta_{con}, \lambda, ta)$

où

$X_{\text{noeud}} = \{(p,v) | p \in IPorts, v \in Xp\}$

$IPort = \{"tache", "faute"\}, avec$ Xtache = Xfaute = job

$Y_{\text{noeud}} = \{(p,v) | p \in Oports, v \in Yp\}$

OPort = {"*resultat*","*erreur*"}, avec Yresultat = Yerreur = job

S = {"*attente*","*active*"}

$\delta_{ext} = (Q$ x tache) :

- si S ="active" ou S ="attente" alors S ="active"

$\delta_{\text{int}}(S)$:

- si S ="active" alors S ="active"

$\delta_{con}(Q$ x $X_{\text{Noeud}})$:

- $\delta_{ext} = (Q$ x tache)
- $\delta_{\text{int}}(S)$

$\lambda(S) = (resultat, job)$ si S ="active"

$\varnothing$ sinon

$ta(S)$:

- σ si S ="active"
- INFINITY si S ="attente"

Figure 3. *Formal description of a correct node*

Where Node = (X Node , Y Node , S, δ ext , δ int , δ con , λ , ta) is the formal description of a node with: X Node = {{task, fault} , job} , Y Node = {{result, error}, job } , S = { waiting, active }, δ ext = {If the node is in a "waiting" state or in an "active" state, and a "job" arrives through the "task" port it remains in the "active" state. No "job" can be allowed to arrive through the "fault" port}. δ int = { during the execution of "jobs", the state of the system remains "active" no matter what happens in the system}. δ con = { if when a "job" that is already in the system wants to be executed and at the same time another "job" arrives through the "task" port, the system executes first the "job" that arrives through the "task" port}. λ = {at the end of the execution of the "job" if the system state was "active", the processed "job" is sent to the "result" output}. ta = {the execution time of each "job" is 100s except when the system is in "waiting" then ta is equal to INFINITY}.

The simulation of the above model shows us the different states of the node in Figure 4.

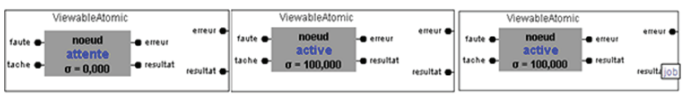


Figure 4. *The states of the correct node during the simulation*

The result of the node simulation is shown in Figure 4 .

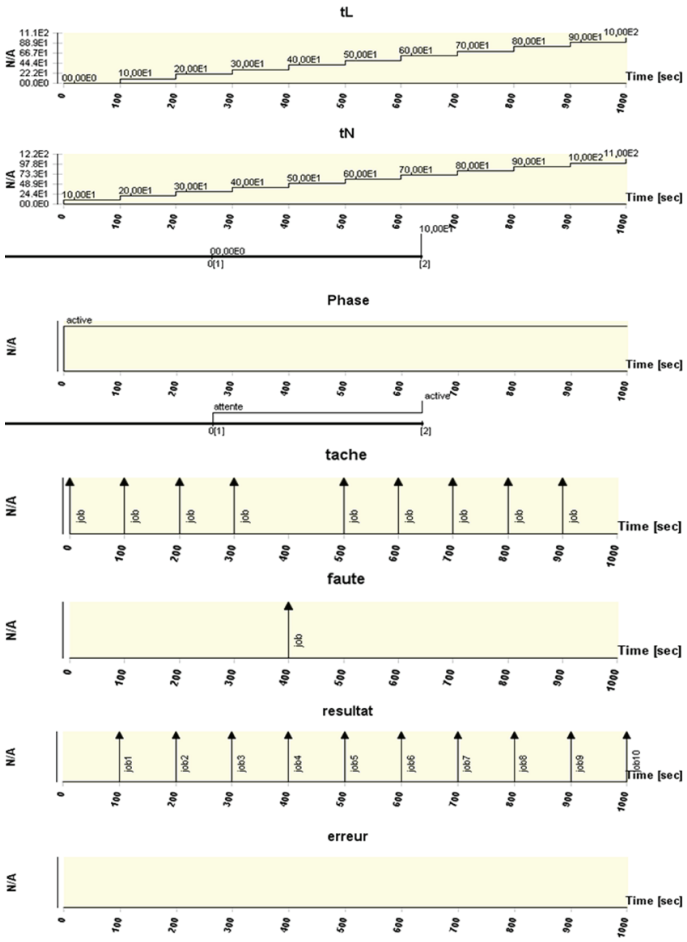


Figure 5. *Simulation of a correct knot*

The simulation in Figure 5 is structured as follows: Figure 5.a, represents the execution time of the last event, i.e. the time of the state being processed. Figure 5.b represents the execution time of the next event. Figure 5.c-1 represents the evolution of the system states during the execution of the jobs and Figure 5.c-2 represents the system state before the arrival of the jobs. Figure 5.d and Figure 5.e represent the two input ports "task" and "fault", when a job is submitted for processing to the grid, it enters through the "task" port and when a fault is committed it enters through the "fault" port . Figure 5.f and Figure 5.g represent the output ports "result" and "error" respectively. When a job is processed successfully, the result is returned via the "result" port and when the system encounters a failure, the result is returned via the "error" port.

In this simulation the jobs are sent by the input port "task", once processed the results are returned by the output port "result". At t=400s we create (inject) a fault but as we simulate a correct node, the node does not generate any error.

Correct Grid Model

In this model, the N model represents the computing grid. Jobs can be submitted via two ports to the grid, "task" and "fault". The submitted jobs arrive at the task manager (input) "WMSI", which is responsible for finding available nodes and sending them parts of the job. Once the jobs are processed by the different nodes, the parts of the job are sent back to the task manager (output) "WMSO" to be reconstructed, restored and sent back to the output port.

$$N = (X, Y, D, \{M_d | d \in D\}, EIC, EOC, IC)$$

où

$$IPort = \{"tache", "faute"\}, avec\ \text{Xtache} = \text{Xfaute} = \text{job}$$

$$X = \{(p, v) | p \in IPorts, v \in Xp\}$$

$$\text{OPort} = \{"resultat", "erreur"\}, \text{avec Yresultat} = \text{Yerreur} = \text{job}$$

$$Y = \{(p, v) | p \in Oports, v \in Yp\}$$

$$\text{D} = \{\text{noeud1}, \text{noeud2}, ..., \text{noeud100}, \text{WMSI}, \text{WMSO}\};;$$

$$M_d = \{M_{noeud1}, M_{noeud2}, ..., M_{noeudn}, M_{WMSI}, M_{WMSO}\};$$

$$EIC = \begin{Bmatrix} (N, "tache"), (WMSI, "tache") \\ (N, "faute"), (WMSI, "faute") \end{Bmatrix}$$

$$IC = \begin{Bmatrix} \begin{Bmatrix} (WMSI, "tache"), (noeud1, "tache") \\ (WMSI, "tache"), (noeud2, "tache") \\ ..... \\ (WMSI, "tache"), (noeud100, "tache") \end{Bmatrix} \\ \begin{Bmatrix} (WMSI, "faute"), (noeud1, "faute") \\ (WMSI, "faute"), (noeud2, "faute") \\ ..... \\ (WMSI, "faute"), (noeud100, "faute") \end{Bmatrix} \\ \begin{Bmatrix} (noeud1, "resultat"), (WMSO, resultat) \\ (noeud2, "resultat"), (WMSO, resultat) \\ ..... \\ (noeud100, "resultat"), (WMSO, resultat) \end{Bmatrix} \\ \begin{Bmatrix} (noeud1, "erreur"), (WMSO, erreur) \\ (noeud2, "erreur"), (WMSO, erreur) \\ ..... \\ (noeud100, "erreur"), (WMSO, erreur) \end{Bmatrix} \end{Bmatrix}$$

$$EOC = \begin{Bmatrix} (WMSO, resultat), (N, resultat) \\ (WMSO, erreur), (N, erreur) \end{Bmatrix}$$

Figure 6. *Simulation of a grid that cannot receive a failure*

The model in Figure 6 is the formal specification of a grid consisting of 100 nodes and a task manager (WMSI and WMSI). The model has two input ports "task" and "fault". The "jobs" are submitted to the grid by the "task" port and when we want to inject a failure into the system it is injected by the "fault" port. The "jobs" submitted to the system first arrive at the task manager which has the name "WMSI" as input. Once the "job" is at the task manager, the WMSI searches for available nodes and sends parts of the "job". In this modeling, we considered that the tasks are perfectly balanced, that is to say that each node executes the tasks during the same time. Once the different parts of the job are executed by the different nodes, the latter are sent back to the output task manager (WMSO). The WMSO rebuilds the job from scratch and if there was no failure the result is returned by the "result" port otherwise the result is returned by the "error" port. The result of the simulation of this model is shown in Figure 7.

Figure 7.a. represents the execution time of the different events that occur in the system. The jobs are injected by the input port "task" every 50s represented here by Figure 7.b and even when the jobs are not injected into the system, the jobs are automatically generated inside the system. We inject three errors into the system Figure 7.c. at t=300s, t=350s and t=400s the system being configured to receive no failures, the system behaves perfectly well Figure 7 .d .

It can be seen that despite the injection of errors, the system does not suffer any failures Figure 7.e.

In the continuation of this work we will now present the behavior of a grid when it receives a frank failure during the execution of tasks.

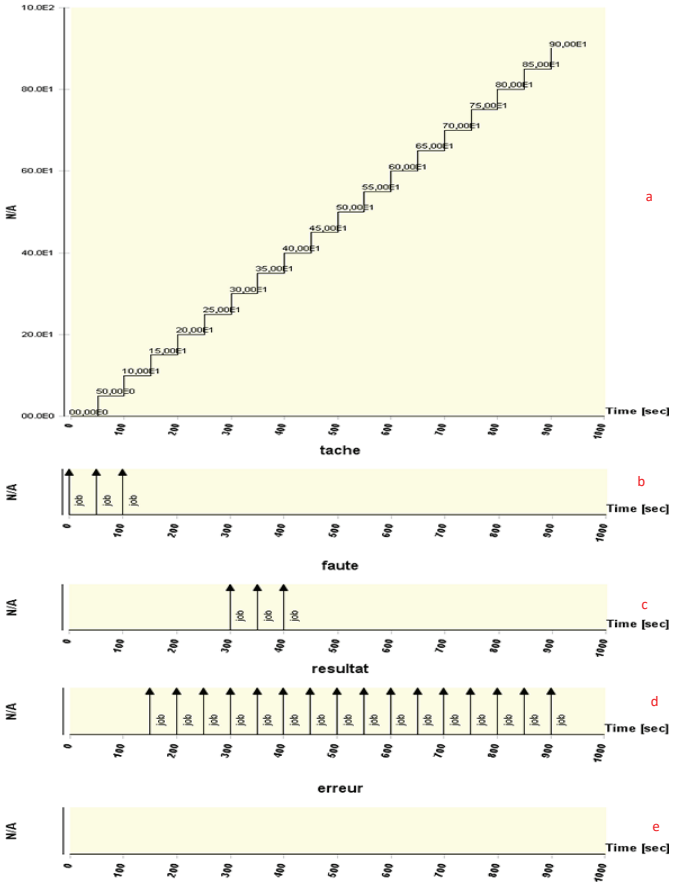


Figure 7.

Modeling of frank failures in a grid

Failure or crash (fail-stop) is a failure during which the component ceases all interaction with the other components of the system . The component behaves according to its specification until it suffers a frank failure. From this point on, it definitively ceases all activity [1]. When it occurs in a computing grid, part of the resource that is processed is totally lost. Frank failures can be caused by:

- Hardware failures, we can have a frank failure because of one of its defective components presented in section 2.2 of this document.
- A natural disaster such as a fire, for example, can also be the cause of a clear failure.
- Ransomware attacks, which are attacks that prevent a user from accessing their data by encrypting the data or simply blocking access to the machine [26].
- Human error is also a common cause of outright failures.
- Software failures also cause many outright failures.

The consequences can be serious for computing grids:

- Loss of turnover: the drop, or even loss, of productivity during the entire outage will immediately reduce turnover.
- Data loss: Data between the last backup and the outright failure is lost.
- Loss of reputation: A computing grid in which data is lost or processing encounters many problems leads to loss of confidence of customers and suppliers [27].

When a frank failure occurs in a node, the node that suffers the failure stops abruptly, and the data that was processed by this node is lost. In the following section we present the modeling of a node when it suffers a frank failure.

Failed node

In this part we present the modeling and simulation of a node that can receive frank failures.

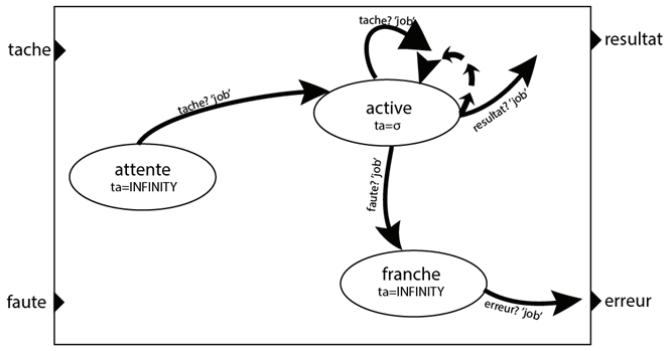


Figure 8: Model of a frank faulty node

The model presented in Figure 9 is almost that of a correct node except that this model has an additional "frank" state. In this model if a fault is committed, the system definitively goes into a frank failure state and there is no way to return to an "active" state of the system. The formal specification of the graphical model of Figure 8 is presented in Figure 9 .

$PDEVS = (X_{Noeud}, Y_{Noeud}, S, \delta_{ext}, \delta_{\text{int}}, \delta_{con}, \lambda, ta)$

où

$X_{\text{noeud}} = \{(p,v) | p \in IPorts, v \in Xp\}$

$IPort = \{"tache","faute"\}, avec$ Xtache = Xfaute = job

$Y_{\text{noeud}} = \{(p,v) | p \in Oports, v \in Yp\}$

OPort = {"*resultat*","*erreur*"},avec Yresultat = Yerreur = job

S = {"*attente*","*active*","*franche*"}

δ_{ext} = (Q x tache) :

- si S ="active" ou S ="attente" alors S ="active"

(Q x faute) :

- si S ="active" alors S ="franche"

$\delta_{\text{int}}(S)$:

- ("*active*") si *S* ="*active*"

$\delta_{con}(Q \text{ x } X_{\text{Noeud}})$:

- $\delta_{ext} = (Q \text{ x } X_{\text{Noeud}})$
- $\delta_{\text{int}}(S)$

$\lambda(S)$ = (*resultat*, *job*) si S ="active"

(erreur, job) sinon

ta(*S*) :

- σ si S ="active"
- *INFINITY* si S ="*franche*"
- *INFINITY* si S ="attente"

Figure 9: *Formal description of the frank failure node*

The simulation Figure 9 represents the behavior of a node that receives a frank failure while it executes tasks. In this model the node can end up in three states unlike the model which represents a correct node. With this model if during the execution of tasks by the node, an error is injected by the "error" port, then a failure occurs and the system goes into a "frank" state. The node therefore remains blocked in the "frank" state and will no longer be able to function.

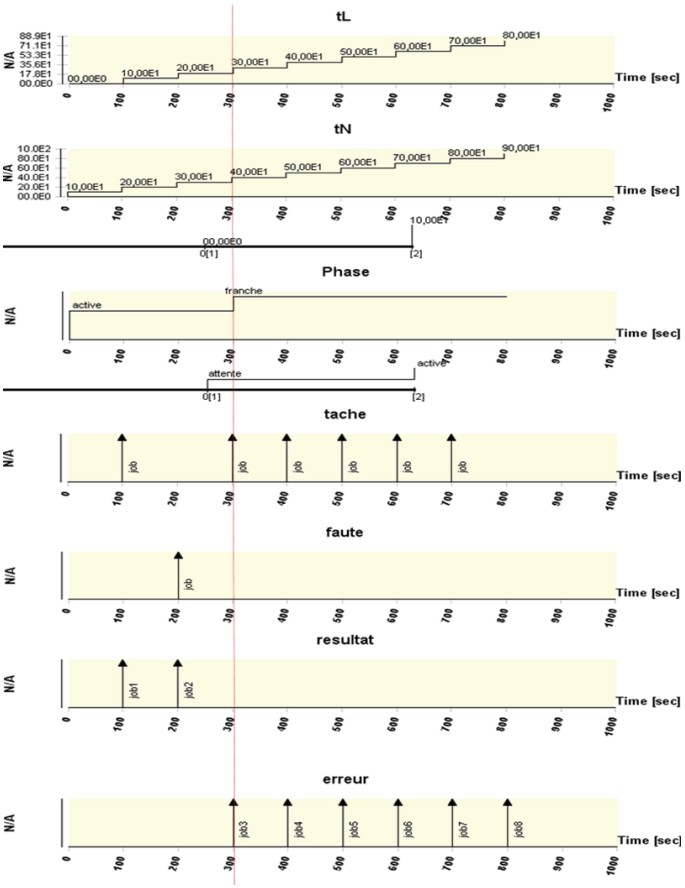


Figure 10: *Simulation of a node subjected to a frank failure*

In this simulation the node behaves as specified until an error is introduced at t=200s, a hard failure occurs and the system enters a hard failure state Figure 10 .

Model of the faulty grid

The model of a node that we present here is a faulty node model, that is, a model that can receive frank failures during the execution of tasks.

The following simulation is of a computing grid whose nodes can receive hard failures. The simulation in Figure 12 represents a grid model that can receive hard failures without a fault tolerance protocol.

In this simulation, all nodes operate normally until a fault occurs on one of the nodes. So from t=200 a failure occurs and this node remains faulty because there is no fault tolerance technique. In this case the node permanently loses the job it was processing. We present in the rest of this work the simulation of a grid that will receive a frank failure. But this grid is equipped with a fault tolerance protocol using an overlap method.

$$N = (X, Y, D, \{M_d \mid d \in D\}, EIC, EOC, IC)$$

où

$$IPort = \{"tache", "faute"\}, avec\ \text{Xtache} = \text{Xfaute} = \text{job}$$

$$X = \{(p, v) \mid p \in IPorts, v \in Xp\}$$

$$\text{OPort} = \{"resultat", "erreur"\}, \text{avec Yresultat} = \text{Yerreur} = \text{job}$$

$$Y = \{(p, v) \mid p \in Oports, v \in Yp\}$$

$$\text{D} = \{\text{noeud1}, \text{noeud2}, ..., \text{noeud100}, \text{WMSI}, \text{WMSO}\};;$$

$$M_d = \{M_{\text{noeud1}}, M_{\text{noeud2}}, ..., M_{\text{noeud}n}, M_{\text{WMSI}}, M_{\text{WMSO}}\};$$

$$EIC = \begin{Bmatrix} (\text{N}, "\text{tache}"), (\text{WMSI}, "\text{tache}") \\ (\text{N}, "\text{faute}"), (\text{WMSI}, "\text{faute}") \end{Bmatrix}$$

$$IC = \begin{Bmatrix} \begin{Bmatrix} (\text{WMSI}, "\text{tache}"), (\text{noeud1}, "\text{tache}") \\ (\text{WMSI}, "\text{tache}"), (\text{noeud2}, "\text{tache}") \\ ..... \\ (\text{WMSI}, "\text{tache}"), (\text{noeud100}, "\text{tache}") \end{Bmatrix} \\ \begin{Bmatrix} (\text{WMSI}, "\text{faute}"), (\text{noeud1}, "\text{faute}") \\ (\text{WMSI}, "\text{faute}"), (\text{noeud2}, "\text{faute}") \\ ..... \\ (\text{WMSI}, "\text{faute}"), (\text{noeud100}, "\text{faute}") \end{Bmatrix} \\ \begin{Bmatrix} (\text{noeud1}, "resultat"), (\text{WMSO}, resultat) \\ (\text{noeud2}, "resultat"), (\text{WMSO}, resultat) \\ ..... \\ (\text{noeud100}, "resultat"), (\text{WMSO}, resultat) \end{Bmatrix} \\ \begin{Bmatrix} (\text{noeud1}, "erreur"), (\text{WMSO}, erreur) \\ (\text{noeud2}, "erreur"), (\text{WMSO}, erreur) \\ ..... \\ (\text{noeud100}, "erreur"), (\text{WMSO}, erreur) \end{Bmatrix} \end{Bmatrix}$$

$$EOC = \begin{Bmatrix} (\text{WMSO}, resultat), (N, resultat) \\ (\text{WMSO}, erreur), (N, erreur) \end{Bmatrix}$$

Figure 11. *Formal description of the grid that can accommodate frank failures.*

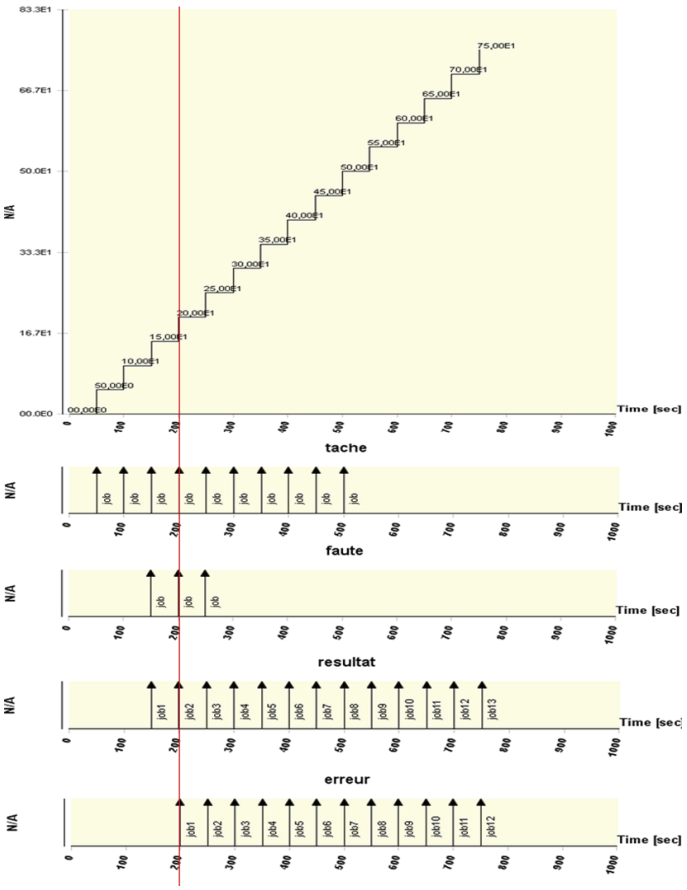


Figure 12. *Simulation d'une grille qui peut recevoir les défaillances franches sans méthode de tolérances aux fautes*

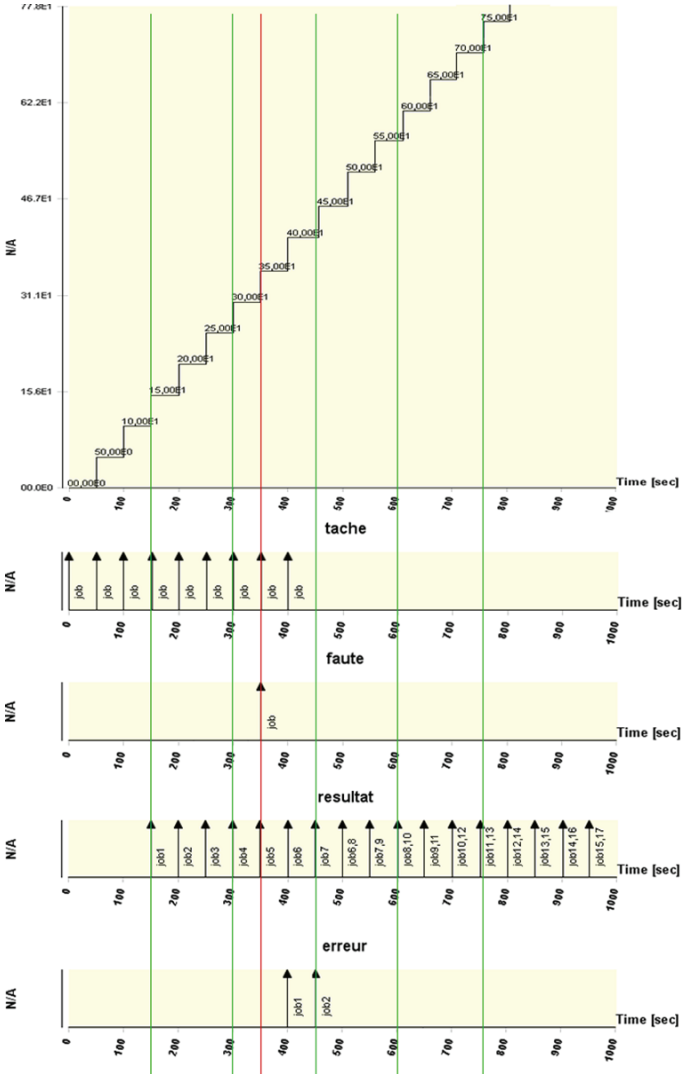


Figure 13. *Simulation of a grid that can accommodate frank failures equipped with a fault tolerance protocol.*

In the simulation presented in Figure 13 , when a clear failure occurs in one of the nodes, the nodes are configured to communicate every 150 seconds (Green bands). When the system finds that one of the nodes is not responding, a fault tolerance technique is applied. Here the fault tolerance method used is "backward recovery" which means that the part of the job that was processed by the failed node is sent to another available node to resume execution of this job. We can therefore see that after the recovery at t=500s while the nodes that have not encountered a failure already return job 8, the node that resumes the job of the failed node returns job 6 at the same time.

Modeling and simulation of anticipation of frank failures

In this part, we present a model for anticipating frank failures in computing grids. The model works as follows: the system starts in a "waiting" state, once a job is sent to the node, the node goes into an "active" state. During processing, if a job enters through the "task" port, the node remains in an "active" state. If during job processing, the fan rotation speed is lower than the minimum rotation speed allowed on the motherboard or if the rotation speed is higher than the maximum rotation speed allowed on the motherboard and other components of the

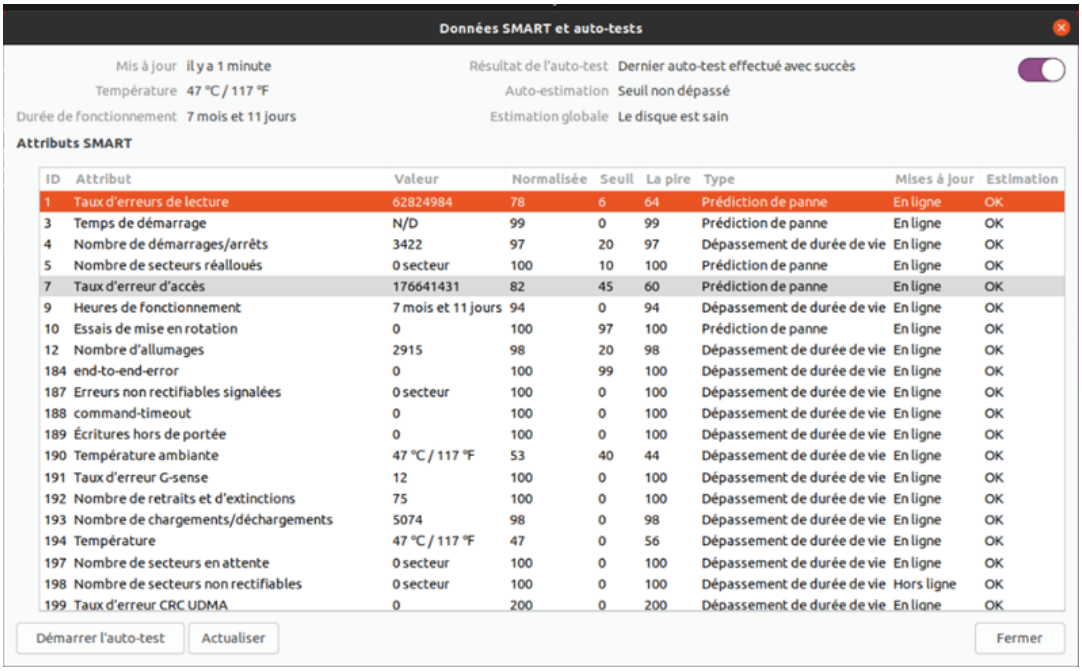


Figure 14. *SMART data from a Lenovo Thinkpad T530*

node, the node reports itself as faulty to the task manager. And the task manager will no longer consider the node thereafter. A function called SMART predicts failures that can happen to the hard disk, so SMART data control can predict frank failures. Once a frank failure is predicted, the system reports to the task manager by error output.

Figure 15 shows us some SMART data from a Lenovo Thinkpad T530. This data allows the monitoring and control of a computer's hard disk and indicates if a failure may affect the computer . The graphical model for predicting hard failures in a computing grid is as follows.

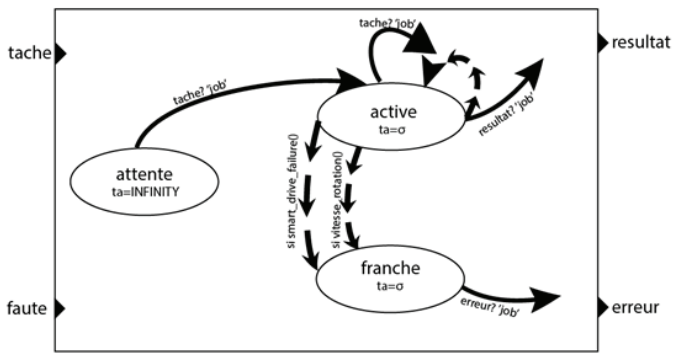


Figure 15. *Model for anticipating frank failures in a node.*

$PDEVS = (X_{Noeud}, Y_{Noeud}, S, \delta_{ext}, \delta_{\text{int}}, \delta_{con}, \lambda, ta)$

où

$X_{\text{noeud}} = \{(p,v) | p \in IPorts, v \in Xp\}$

$IPort = \{"tache", "faute"\}, avec$ Xtache = Xfaute = job

$Y_{\text{noeud}} = \{(p,v) | p \in Oports, v \in Yp\}$

OPort = {"*resultat*","*erreur*"},avec Yresultat = Yerreur = job

S = {"*attente*","*active*","*franche*"}

$\delta_{ext} = (Q$ x tache) :

- si S ="active" ou S ="attente" alors S ="active"

$\delta_{\text{int}}(S)$:

- ("*franche*") si smart_drive_faillure() ou vitesse_rotation()
- ("*active*") sinon

$\delta_{con}(Q \text{ x } X_{\text{Noeud}})$:

- $\delta_{\text{int}}(S)$
- $\delta_{ext} = (Q \text{ x } X_{\text{Noeud}})$

$\lambda(S) = (resultat, job)$ si S ="active"

$\varnothing$ sinon

$ta(S)$:

- σ si S ="active"
- σ si S =" *franche*"
- *INFINITY* si S ="attente"

Figure 16. *Formal description of anticipation of frank failures in a node.*

The formal specification of the graphical model of the node in Figure 15 is given below in Figure 16 .

The anticipation model proposed above is done at the node level. Once the smart_drive_failure() function has detected a hard disk fault or the rotation_speed() function has detected a fan fault, the detected faults will be sent to the middleware's MDS (Monitoring and Discovering Service). The task manager will therefore consult the MDS to obtain information on the different resources and identify those that are likely to have a failure. The simulation of this model is presented in Figure 17.

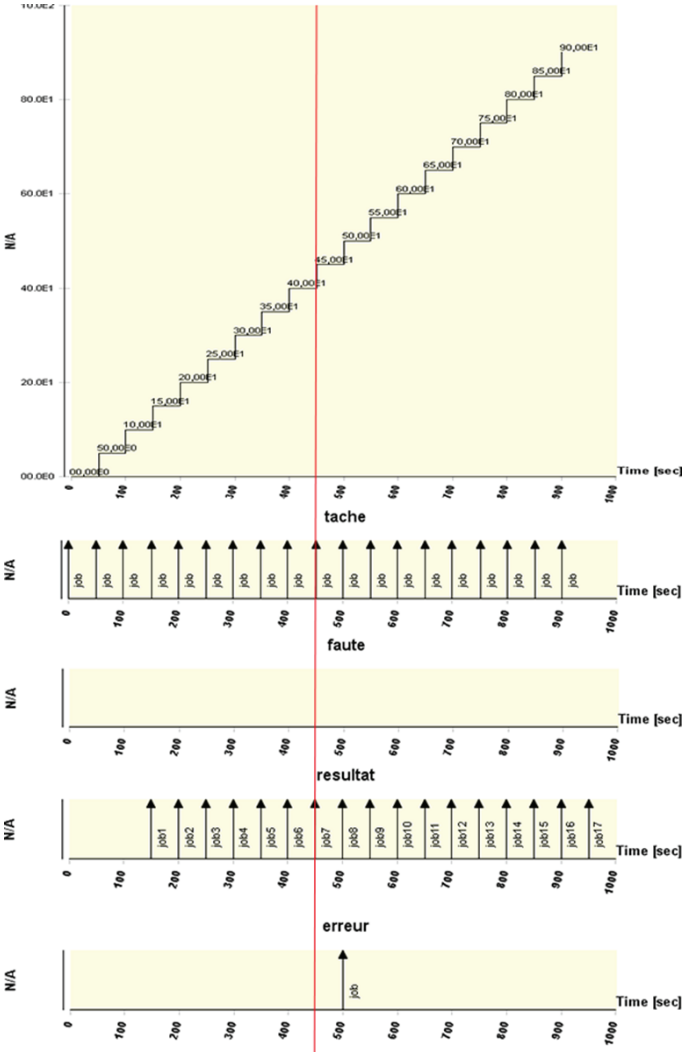


Figure 17. Simulation of a grid that anticipates frank failures .

The simulation of Figure 19 above shows that of a model that anticipates frank failures. When one of the signs of a frank failure occurs, the task manager lets the node finish the task currently running and the node will not receive any more tasks for the future. In our case a probable failure is detected at t=450 because the rotation speed of the fan of a node is below its minimum speed recommended by the manufacturer.

The frank failure anticipation method proposed in this paper has been evaluated and compared with existing fault tolerance methods (Replication and Backward Recovery Tolerance Method) and the result is shown in Figure 20 .

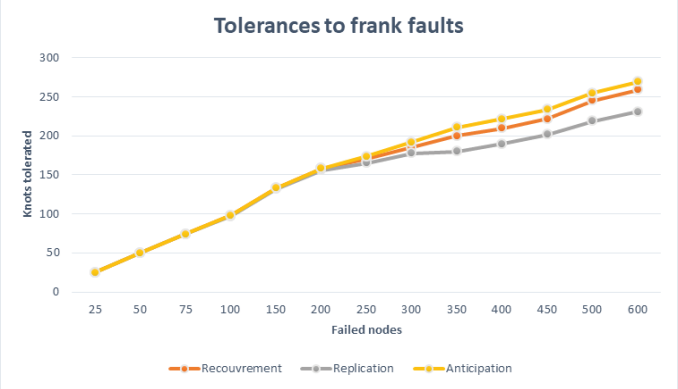


Figure 18. Simulation of a grid that anticipates frank failures .

We simulated a grid of 600 nodes and each time we increased the number of failed nodes to study the fault tolerance capacity of the replication, recovery and anticipation methods. The results presented in Figure 22 show us that the anticipation model tolerates frank failures better than the recovery and anticipation models starting from 41.66% of failed nodes in the grid. Before reaching the rate of 41.66% of failed nodes in the grid, the three methods behave almost identically in terms of the number of failed nodes tolerated.

Conclusion

In this paper, we have presented the models for anticipating frank failures in computing grids. The failure anticipation models that we have proposed are based on the study of the different components of the grid nodes. We have therefore proposed at the beginning fault tolerance models based on the standard fault tolerance models that are: replication and back-recovery. Then we have proposed a fault tolerance model that monitors the different components of the nodes to anticipate probable frank failures. We have obtained the best results when the number of failing nodes to tolerate starts to be greater than 200 nodes.

References

1. Monnet S. Gestion des données dans les grilles de calcul : support pour la tolérance aux fautes et la cohérence des données. RENNES: UNIVERSITÉ DE RENNES 1; 2006.
2. Ndiaye NM. Techniques de gestion des défaillances dans les grilles informatiques tolérantes aux fautes. Paris VI: UNIVERSITÉ PIERRE ET MARIE CURIE; 2013.
3. Stellner G. CoCheck: Checkpointing and process migration for MPI. International Parallel Processing Symposium; Honolulu, 1996.
4. Bosilca G, Bouteiller A, Cappello F, et al. MPICH-V: Toward a scalable fault tolerant MPI for volatile nodes. Supercomputing, ACM/IEEE 2002 Conference; Paris, 2002.
5. Jafar S. Programmation des systèmes parallèles distribués : tolérance aux pannes, résilience et adaptabilité. GRENOBLE: INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE; 2006.
6. Hursey J, Mattox T, Lumsdaine A. The design and implementation of checkpoint/restart process fault tolerance for Open MPI. Proceedings of the 21st IEEE International Parallel and Distributed Processing Symposium (IPDPS). IEEE; 2007.
7. Rebbah M. Tolérance aux fautes dans les grilles de calcul. Oran:

Université des Sciences et de la Technologie d'Oran Mohamed Boudiaf; 2015.

8. Robert T. Tolérance aux Fautes des Systèmes Informatiques. Paris: Telecom Paristech; 2006.
9. Bennani T. Tolérance aux fautes dans les systèmes répartis à base d'intergiciels réflexifs standards. Toulouse: Institut National des Sciences Appliquées de Toulouse; 2005.
10. Castro M, Liskov B. Practical Byzantine Fault Tolerance. Proceedings of the Third Symposium on Operating Systems Design and Implementation; New Orleans, 1999.
11. Abd-El-Malek M, Ganger G, Goodson G, Reiter M, Wylie J. Fault-Scalable Byzantine Fault-Tolerant Services. ACM SIGOPS Operating Systems Review. 2005;39(5):59-74.
12. Kotla R, Alvisi L, Dahlin M, Clement A, Wong E. Zyzzyva: Speculative Byzantine fault tolerance. ACM Transactions on Computer Systems. 2010;27(4):1-39.
13. Guerraoui R, Schiper A. Fault-Tolerance by Replication in Distributed Systems. International Conference on Reliable Software Technologies; Lausanne, 1996.
14. Buyya R, Venugopal S. A Gentle Introduction to Grid Computing and Technologies. Computer Society of India. 2005;1(29):9-19.
15. CNRS. Grille de calcul : l'internet du calcul intensif. IN2P3; 2015. Available at: http://old.in2p3.fr/presentation/thematiques/grille/grille.htm. Accessed October 10, 2023.
16. Lopez A, Sacquin-Mora S, Dimitrova V, et al. Protein-protein interactions in a crowded environment: an analysis via cross-docking simulations and evolutionary information. PLoS Computational Biology. 2013;9(12):1-18.
17. Lapirot O. Grille informatique : l'union fait la force de calcul. 01net; November 4, 2010. Available at: https://www.01net.com/actualites/grille-informatique-lunion-fait-la-force-de-calcul-523427.html. Accessed January 2, 2024.
18. Chervenak A, Deelman E, Kesselman C, et al. High-Performance Remote Access to Climate Simulation Data: A Challenge Problem for Data Grid Technologies. Supercomputing, ACM/IEEE 2001 Conference; 2001.
19. Lee C, Kesselman C, Schwab S. Near-real-time satellite image processing: Metacomputing in CC++. IEEE Computer Graphics and Applications. 1996;16(4):79-84.
20. Adela LD. Un guide complet sur la façon de connaître les signes d'une panne de disque dur. fonedog; March 29, 2021. Available at: https://www.fonedog.fr/data-recovery/signs-of-hard-drive-failure.html.
21. Foster I, Kesselman C. The Grid: Blueprint for a New Computing Infrastructure. Morgan Kaufmann; 2003.
22. Avizienis A, Laprie JC, Randell B. Fundamental Concepts of Dependability. Newcastle: Newcastle University; 2001.
23. Zeigler BP, Kim TG, Praehofer H. Theory of Modeling and Simulation. Orlando: Academic Press; 2000.
24. Yougouda RA, Nkenlifack M, Kamla VC, Bitjoka L. Model for Anticipating Failures by Omission in Calculation Grids. Open Journal of Optimization. 2021;10(3):71-87.
25. Zeigle BP, Sarjoughian HS. Introduction to DEVS Modeling and Simulation with JAVA: Developing Component-Based Simulation Models. Arizona: Arizona Center for Integrative Modeling and Simulation; 2005.
26. Misini L. Etude des Ransomware : Vecteur d'attaque, fonctionnement, économie et réponse légale. Genève: HEG-GE; 2017.
27. Florian. Quelques causes et conséquences des crashs informatiques. Alliance Informatique; 2018. Available at: https://www.alliance-informatique.fr/revue-blog/quelles-sont-les-causes-et-les-consequences-dun-crash-informatiq. Accessed January 12, 2024.
28. Ndiaye NM. Techniques de gestion des défaillances dans les grilles informatiques tolérantes aux fautes. Paris: Université Pierre et Marie Curie; 2013.